better BASIC for the apple®

J.N.P. Hume
R.C. Holt

BETTER BASIC
for the
APPLE

BETTER BASIC for the APPLE

J.N.P. Hume

R.C. Holt

Department of Computer Science
University of Toronto

RESTON PUBLISHING COMPANY, INC., Reston, Virginia
A Prentice-Hall Company

ISBN 0-8359-0466-0

© 1983 by Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia 22090

All rights reserved. No part of this book may be reproduced, in any way or by any means, without permission in writing from the publisher.

Printed in the United States of America

CONTENTS

1. INTRODUCTION TO STRUCTURED PROGRAMMING 1

WHAT IS PROGRAMMING? 1
WHAT IS STRUCTURED PROGRAMMING? 2
WHAT IS BASIC? 3
CORRECTNESS OF PROGRAMS 4
WHAT IS APPLESOFT II BASIC? 5
SUMMARY 6

2. THE COMPUTER 9

PARTS THAT MAKE THE WHOLE 9
CODED INFORMATION 10
MEMORY 11
ARITHMETIC UNIT 13
CONTROL UNIT 14
INPUT AND OUTPUT 15
PROGRAM TRANSLATION 16
SUMMARY 17

3. GETTING THE COMPUTER TO CALCULATE AND OUTPUT 19

CHARACTERS 19
NUMBERS 21
CHARACTER STRINGS 22
EXPRESSIONS 22
EXAMPLES OF ARITHMETIC EXPRESSIONS 23
OUTPUT 24
THE PROGRAM 26
ANOTHER PROGRAM 27
PRINTED OUTPUT 27
SUMMARY 28
EXERCISES 30

4. EDITING AND SAVING PROGRAMS 33

INITIALIZING YOUR OWN DISKETTE 33
SAVING A PROGRAM 34
EDITING A PROGRAM 36
COMMANDS THAT CHANGE FILES 38
COPYING DISKETTES 39
SUMMARY 39
EXERCISES 42

5. VARIABLES, CONSTANTS, AND ASSIGNMENTS 45

VARIABLES 45
ASSIGNMENT STATEMENTS 47
TRACING EXECUTION 49
INPUT OF DATA 50
AN ALTERNATIVE WAY TO INPUT DATA 52
REMARKS 52
AN EXAMPLE PROGRAM 53
LABELLING OF OUTPUT 55
PROGRAM TESTING 57
COMMON ERRORS IN PROGRAMS 58
COMMUNICATING WITH THE BASIC INTERPRETER 59
SUMMARY 60
EXERCISES 62

6. CONTROL FLOW 65

COUNTED LOOPS 65
CONDITIONS 67
CONDITIONAL LOOPS 68
UNCONDITIONAL LOOP 70
READING INPUT 70
EXAMPLES OF LOOPS 72
BRANCHES IN CONTROL FLOW 75
THREE-WAY BRANCHES 77
EXAMPLES OF SELECTION 79
MULTI-WAY BRANCHES 80
PARAGRAPHING THE PROGRAM 81
TRACING CONTROL FLOW 82
SUMMARY 83
EXERCISES 87

7. STRUCTURING CONTROL FLOW 91

STRUCTURE OF LOOPS 91
FLOW CHARTS 93
PROBLEMS WITH LOOPS 95
NESTED LOOPS 95
AN EXAMPLE PROGRAM 96
LOOPS WITH MULTIPLE CONDITIONS 98
IF STATEMENTS WITH MULTIPLE CONDITIONS 100
SUMMARY 101
EXERCISES 101

8. ARRAYS 105

DECLARATION OF ARRAYS 105
TWO-DIMENSIONAL ARRAYS 107
AN EXAMPLE PROGRAM 108
ARRAYS AS DATA STRUCTURES 110
OTHER DATA STRUCTURES 111
SUMMARY 112
EXERCISES 112

9. ALPHABETIC INFORMATION HANDLING 115

CHARACTER STRINGS 115
READING AND OUTPUTTING CHARACTERS 116
SEQUENCING STRINGS 119
JOINING STRINGS TOGETHER 120
SELECTING PARTS OF STRINGS 121
AN EXAMPLE PROGRAM 123
CHARACTER CODES 124
SUMMARY 126
EXERCISES 128

10. STRUCTURING YOUR ATTACK ON THE PROBLEM 131

STEP-BY-STEP REFINEMENT 131
TREE STRUCTURE TO PROBLEM SOLUTION 132
CHOOSING DATA STRUCTURES 133
GROWING THE SOLUTION TREE 133
DEVELOPING AN ALGORITHM 134
THE COMPLETE PROGRAM 135

ASSESSING EFFICIENCY 136
A BETTER ALGORITHM 137
BETTER ALGORITHMS 139
SUMMARY 140
EXERCISES 141

11. THE COMPUTER CAN READ ENGLISH 145

WORD RECOGNITION 146
WORD STATISTICS 148
READING BASIC 150
SUMMARY AND EXERCISES 151

12. SUBPROGRAMS 153

SUBROUTINES 153
PROBLEMS WITH SUBROUTINES 156
FUNCTIONS 157
BUILT-IN FUNCTIONS 158
RANDOM NUMBERS 159
SUMMARY 159
EXERCISES 162

13. GRAPHICS 163

TEXT AND GRAPH MODES 163
CHOOSING A COLOR FOR LOW-RESOLUTION GRAPHICS 164
PLOTTING A POINT ON THE SCREEN 165
DRAWING LINES ON THE SCREEN 166
GAME CONTROLS AND GRAPHICS 168
MAKING SOUNDS ON THE COMPUTER 169
ANIMATION WITH GRAPHICS 170
HIGH-RESOLUTION GRAPHICS 171
PLOTTING LINES FROM POINT TO POINT 172
PLOTTING GRAPHS 173
SUMMARY 176
EXERCISES 178

14. SEARCHING AND SORTING 179

LINEAR SEARCH 179
TIME TAKEN FOR SEARCH 181

BINARY SEARCH 182
A SUBROUTINE FOR BINARY SEARCH 182
SEARCHING BY ADDRESS CALCULATION 184
SORTING 185
SORTING BY MERGING 186
EFFICIENCY OF SORTING METHODS 187
SUMMARY 188
EXERCISES 189

15. MAKING SURE THE PROGRAM WORKS 191

SOLVING THE RIGHT PROBLEM 191
DEFENSIVE PROGRAMMING 192
ATTITUDE AND WORK HABITS 192
PROGRAM CORRECTNESS 193
PROGRAMMING STYLE 193
USE OF REMARKS AND IDENTIFIERS 193
TESTING 195
DEBUGGING 197
SUMMARY 199
EXERCISES 200

16. FILES AND RECORDS 201

RECORDS 201
FILES IN SECONDARY MEMORY 204
PREPARATION OF FILES 206
LISTING OF FILES 207
MONITORING FILE ACTIONS 207
FILE MAINTENANCE 208
RANDOM ACCESS TO FILES 210
SYSTEM COMMANDS FOR TEXT FILES 211
TEXT FILES FOR EXECUTION 212
SAVING PROGRAMS AS TEXT FILES 213
USE OF END-OF-FILE ERROR 215
SUMMARY 217
EXERCISES 220

17. DATA STRUCTURES 223

LINKED LISTS 223
INSERTING INTO A LINKED LIST 224

MEMORY MANAGEMENT WITH LISTS 225
SUBROUTINE FOR INSERTING INTO A LINKED LIST 226
DELETING FROM A LINKED LIST 229
MAINTAINING A LINKED LIST 230
STACKS 231
A PROGRAM USING A STACK 232
QUEUES 233
TREES 235
ADDING TO A TREE 237
DELETING FROM A TREE 237
LISTING A TREE IN ORDER 238
SUMMARY 239
EXERCISES 240

18. SCIENTIFIC CALCULATIONS 243

EVALUATING FORMULAS 244
PREDECLARED MATHEMATICAL FUNCTIONS 245
GRAPHING A FUNCTION 246
A SUBROUTINE FOR PLOTTING GRAPHS 247
USING THE GRAPH SUBROUTINE 248
FITTING A CURVE TO A SET OF POINTS 249
SOLVING POLYNOMIAL EQUATIONS 251
SOLVING LINEAR EQUATIONS 252
COMPUTING AREAS 253
SUMMARY 254
EXERCISES 256

APPENDIX 1: SYNTAX OF BETTER BASIC FOR THE APPLE 257
APPENDIX 2: APPLESOFT BASIC COMMANDS (DOS COMMANDS) 268

APPENDIX 3: RESERVED WORDS IN APPLESOFT BASIC 270

APPENDIX 4: BUILT-IN APPLESOFT BASIC FUNCTIONS 271

APPENDIX 5: ERROR MESSAGES AND CODES 273

APPENDIX 6: ASCII CODE 275

INDEX 277

PREFACE

The Basic programming language and operating system was devised over twenty years ago. Since that time many improvements have been made in computer programming languages. Basic itself has undergone changes since it was introduced and now it is possible to write Basic programs that reflect the advances in our thinking about how programs ought to be constructed. We must *not* continue to use Basic in an old-fashioned way. We must write better Basic programs. These better programs should conform to all the good qualities associated with the term structured programming.

This book is intended to be read by someone beginning to learn about computer programming. No particular mathematical background beyond arithmetic is needed; examples are taken largely from everyday life. It is our strong conviction that the foundation of computer programming must be carefully laid. Bad habits once begun are hard to change. Computer programming provides excellent experience in the systematic analysis of problems from the statement of *what is to be done* to the final algorithm for *doing it.* This can be very helpful in encouraging logical thinking.

The particular version of Basic presented here is Applesoft II Basic for the Apple computer. It contains a number of extensions to standard Basic, particularly ones to handle graphics and character strings.

In this book, Applesoft Basic is introduced in easy to understand stages. The book is about structured programming and that is what we hope you will be learning by following this step-by-step presentation.

Just as a program provides a list of instructions to the computer to achieve some well-defined goal, the methodology of structured programming provides a list of instructions to persons who write programs to achieve well-defined goals. The goals of structured programming are to get a programming job done correctly and in such a form that later modifications can be done easily. This means that programs must be understood by people other than their authors.

Structured programming is especially important when working on larger programs; a detailed discussion of the techniques of modular programming and top-down design accompanies the introduction of Basic subprograms. Many examples in the book are from data processing, and how to handle disk files and records is discussed at length.

At all times we have tried to present things in easy to understand stages, offering a large number of program examples and exercises to be done on your computer. Each chapter has a summary of the important concepts introduced in it.

This book was prepared using a type setting system on a computer. Each program was tested on the Apple computer by Harriet Hume who was of great assistance during all phases of the book's preparation from initial draft to camera-ready copy. The job of transcribing the authors' pencil scrawls into the computer and supervising the computer type setting process was done with great care and patience by Inge Weber.

It is clear that the Basic programming language will be around for a long time to come. The moment has arrived to switch to better Basic programming. Why settle for less?

J.N.P. Hume

R.C. Holt

BETTER BASIC for the APPLE

Chapter 1

INTRODUCTION TO STRUCTURED PROGRAMMING

We hope that it is no secret that the book has to do with *computers* and particularly with the *use* of computers rather than their design or construction. To use computers you must learn how to speak their *language* or a language that they can understand. We do not actually speak to computers yet, although we may some day; we write messages to them. The reason we write these messages is to instruct the computer about some work we would like it to do for us. And that brings us to *programming.*

WHAT IS PROGRAMMING?

Programming is writing instructions for a computer in a language that it can understand so that it can do something for you. You will be learning to write programs in one particular *programming language* called Basic. When these instructions are entered into a computer directly by means of a keyboard input terminal they go into the part of the computer called its *memory* and are recorded there for as long as they are needed. The instructions could then be *executed* if they were in the language the computer understands directly, the language called machine language. If they are in another language such as Basic they must first be interpreted from the original or *source* program. Each instruction in Basic is interpreted by the interpreter before it can be executed.

Computers can really only do a very small number of different basic things. For example, an instruction which says, STAND ON YOUR HEAD, will get you nowhere. The repertoire of instructions that any computer understands usually includes the ability to move numbers from one place to another in its memory, to add, subtract, multiply, and divide. They can, in short, do all kinds of *arithmetic calculations* and they can do these operations at rates of up to a million a second. Computers are extremely fast calculating machines. But they can do more; they can also handle alphabetic information, both moving it around in their memory and

comparing different pieces of information to see if they are the same. To include both numbers and alphabetic information we say that computers are *data processors* or more generally *information processors.*

When we write programs we write a sequence of instructions that we want executed one after another. But you can see that the computer could execute our programs very rapidly if each instruction were executed only once. A program of a thousand instructions might take only a thousandth of a second. One of the instructions we can include in our programs is an instruction which causes the use of other instructions to be repeated over and over. In this way the computer is capable of repetitious work; it tirelessly executes the same set of instructions again and again. Naturally the data that it is operating on must change with each repetition or it would accomplish nothing.

Perhaps you have heard also that computers can make *decisions.* In a sense they can. These so-called decisions are fairly simple. The instructions read something like this:

```
IF JOHN IS OVER 16 THEN PLACE HIM ON THE HOCKEY TEAM
      ELSE PLACE HIM ON THE SOCCER TEAM
```

Depending on the *condition* of John's age, the computer could place his name on one or other of two different sports teams. It can *decide* which one if you tell it the decision criterion, in our example being over sixteen or not.

Perhaps these first few hints will give you a clue to what programming is about.

WHAT IS STRUCTURED PROGRAMMING?

The term *structured programming* is used to describe a number of techniques for writing *better* programs. Just as programs provide a list of instructions to the computer to achieve some well-defined goal, structured programming provides a list of instructions to persons who write programs to achieve some well-defined goals. The goals of structured programming are, first, to get the job done. This deals with *how* to get the job done and how to get it done *correctly.* The second goal is concerned with having it done so that other people can see how it is done, both for their education and in case these other people later have to make changes in the original programs.

Computer programs can be very simple and straightforward but many applications require that very large programs be written. The very size of these programs makes them complicated and difficult to understand. But if

they are well-structured, then the complexity can be controlled. Controlling complexity can be accomplished in many different ways and all of these are of interest in the cause of structured programming. The fact that structured programming is the "new philosophy" encourages us to keep track of everything that will help us to be better programmers. We will be cataloguing many of the elements of structured programming as we go along, but first we must look at the particular programming language you will learn.

WHAT IS BASIC?

Basic is a language that has been developed to be independent of the particular computer on which it is run and oriented to the problems that persons might want done. We say that Basic is a *high-level language* because it was designed to be relatively easy to learn. As a problem-oriented language it is concerned with problems of numerical calculations such as occur in scientific and engineering applications as well as with alphabetic information handling required by business and humanities applications. It is the most commonly used language on personal computers.

Basic is a reasonably extensive language, so that although each part is easy to learn, it requires considerable study to master. Many different computers, ranging in size from large computers to microcomputers, have available the facilities to accept programs written in Basic. This means that they have a *Basic interpreter* that will interpret programs written in Basic into the language of the particular machine that they have.

It has been the experience over the past years that a high-level language lasts much longer than machine languages, which change every five years or so. This is because once an investment has been made in programs for a range of applications, you do not want to have to reprogram when a new computer is acquired. What is needed is a new interpreter for the high-level language and all the old programs can be reused.

Because of the long life-span of programs in high-level languages it becomes more and more important that they can be adapted to changes in the application rather than completely reconstructed.

A high-level language has the advantage that well-constructed and well-documented programs in the language can be readily modified. Our aim is to teach you how to write such programs. Basic as a language was developed long before the ideas of structured programming were developed. Most of the programs that are available in Basic are difficult to understand and to modify, especially if they are long programs. For this reason we believe that it is a good idea to start writing Basic programs in a better way. That is why we called the book *Better Basic for the Apple.*

Fortunately the Basic language itself has been extended for the Apple Computer in various ways to make it easier to write better programs. But it has not got a few of the very important features of some of the more recent high-level languages like Pascal. This means that some of the parts of our better Basic programs may seem a bit awkward, more so if you have already learned Basic without thinking about well-structured programs.

There is no substitute for practice in learning to program, so as soon as possible and as often as possible, submit your knowledge to the test by creating your own programs and running them on your computer.

CORRECTNESS OF PROGRAMS

One of the maddening things about computers is that they do exactly what you tell them to do rather than what you want them to do. To get correct results your program has to be correct. When an answer is displayed by a computer you must know whether or not it is correct. You cannot assume, as people often do, that because it was given by a computer it must be right. It is the right answer for the particular program and data you provided because computers now are really very reliable and rarely make mistakes. But is your program correct? Are your input data correct?

One way of checking whether any particular answer is correct is to get the answer by some other means and compare it with the displayed answer. This means that you must work out the answer by hand, perhaps using a hand calculator to help you. When you do work by hand you probably do not concentrate on exactly how you are getting the answer but you know you are correct (assuming you do not make foolish errors). But this seems rather pointless. You wanted the computer to do some work for you to save you the effort and now you must do the work anyway to test whether your computer program is correct. Where is the benefit of all this? The labor saving comes when you get the computer to use your program to work out a similar problem for you. For example, a program to compute telephone bills can be checked for correctness by comparing the results with hand computation for a number of representative customers and then it can be used on millions of others without detailed checking. What we are checking is the method of the calculation.

We must be sure that our representative sample of test cases includes all the various exceptional circumstances that can occur in practice, and this is a great difficulty. Suppose that there were five different things that could be exceptional about a telephone customer. A single customer might have any number of exceptional features simultaneously. So the number of different types of customers might be 32, ranging from those with no

exceptional features to those with all five. To test all these combinations takes a lot of time, so usually, we test only a few of the combinations and hope all is well.

Because exhaustive testing of all possible cases to be handled by a program is too large a job, many programs are not thoroughly tested and ultimately give incorrect results when an unusual combination of circumstances is encountered in practice. You must try to test your programs as well as possible and at the same time realize that with large programs the job becomes very difficult. This has led many computer scientists to advocate the need to *prove* programs correct by various techniques other than exhaustive testing. These techniques rely partly on reading and studying the program to make sure it directs the computer to do the right calculation. Certainly the well-structured program will be easier to prove correct.

WHAT IS APPLESOFT II BASIC?

APPLESOFT II Basic is a version of Basic that was designed particularly for the Apple computer. It incorporates a number of extensions to the Basic language, extensions that allow you to manipulate graphic images and strings of alphabetic information more easily than you can with standard Basic.

In order that the Apple computer can accept programs in Basic and run them it must already have programs inside it. These are called *systems programs.* One of these systems programs is a Basic Interpreter. Another is called its *disk operating system* (DOS). The operating system permits you to enter your program through the keyboard, make any alterations in it that are necessary (*edit* it), store it away in the secondary memory of the computer (*file* it), initiate execution (*run* programs), initiate printed output (prepare *hardcopy*), and so on.

In order to get the APPLESOFT II Basic operating system to accept, edit, file, or execute your Basic program you must give it instructions through the keyboard. These are called *system commands.* So, in addition to learning the Basic programming language for writing Basic programs, you must also learn the APPLESOFT II Basic *command language.* This command language, like the Basic programming language, is extensive and we will be introducing only as much as you will need at any time. It is a good idea to learn a few commands at a time and become confident in their use before learning more.

CHAPTER 1 SUMMARY

The purpose of this book is to introduce computer programming. We have begun in this chapter by presenting the following programming terminology.

Program (or computer program) - a list of instructions for a computer to follow. We say the computer *executes* instructions.

Programming - writing instructions telling a computer to perform certain data manipulations.

Programming language - a language used to write programs that direct the computer to do work for us.

Basic - a popular programming language developed at Dartmouth College under the direction of John G. Kemeny. PL/1, Fortran, Cobol, Pascal, and APL are some other popular programming languages.

APPLESOFT II Basic - a version of Basic that contains extensions to the standard Basic that permit easy manipulation of graphic images and strings of characters. These extensions were devised for the Apple Computer.

High-level language - a programming language that is designed to be convenient for writing programs. Basic is a high-level language.

Interpreter - a systems program that translates a program written in a high-level language, such as Basic, one instruction at a time into a language that can be executed on a computer.

Structured programming - a method of programming that helps us write correct programs that can be understood by others. The Basic language has not been designed to encourage structured programming but, by using the extensions of Applesoft Basic and being careful about how we use Basic, reasonably good programs can be written. This book presents a structured approach to programming.

Correctness of programs - the validity of programs should be checked. This can be attempted by comparing test results produced by the computer with the results of calculations made in another way, for example, by using a hand calculator. Although the ideal is to try to prove a program correct by mathematical means, it is often extremely helpful to read and study the program to see that your intentions will be carried out.

Operating system - a program that is kept in the computer during operation and which permits users to operate. It lets them submit programs, edit them, file them, and execute them. The operating system often is stored on a diskette which is read into the main memory of

the computer when you begin a session at the keyboard.

Command language - a set of commands that cause the operating system to perform certain functions for you. These are used to submit, edit, file, and execute programs.

APPLESOFT II command language - the particular set of commands that are used with the Basic operating system on the Apple II computer.

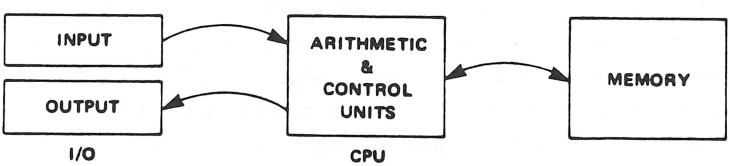
Chapter 2

THE COMPUTER

Before we start writing programs in Basic we will get to know a little bit about computers and how they are organized. A computer is a complex object composed of wires, silicon chips with electronic circuits on them, and so on, but we will not be trying to follow circuit diagrams and worrying about how to build a computer. What we will be interested in is the various main parts of a computer and what the function of each is. In this way your programming will be more intelligent; you will understand a little of what is going on inside the computer.

PARTS THAT MAKE THE WHOLE

We have already mentioned a number of things about computers. They have a *memory* where programs, numbers, and alphabetic information can be recorded. They can add, subtract, multiply, and divide. This means they have a part called the *arithmetic unit.* They can read information from a keyboard input and output results on a screen. They may also have a printer. The printer may output a whole line at a time but it is more common to have a printer that outputs just one character at a time, like a typewriter. We say they have an *input* (for example, keyboard) and an *output* (a screen). The input-output unit is often referred to as the I/O. Computers execute instructions in sequence. The part of the machine that does this is called the *control unit.* The arithmetic unit and the control unit are usually grouped together in a computer and called the *central processing unit* or CPU. So then the computer is thought of as having three parts, memory, I/O and CPU.



MAIN PARTS OF A COMPUTER

We will look at these different parts in turn and see how they work but first we must see how numbers and alphabetic information can be represented in a computer.

CODED INFORMATION

You are probably familiar with the way that information used to travel over telegraph wires in the form of Morse Code. Perhaps you know that each letter or number is coded as a pattern of dots and dashes. For example, the letter A is a dot followed by a dash, E is one dot, V is three dots and a dash. The letters are separated from each other by a pause with no dots or dashes. The famous signal SOS is

. . . _ _ _ . . .

This is an easy one to remember in emergencies. The Morse Code was designed so that the signal could activate some noise-making device and the listener could then translate the coded message back into letters. Modern teletype machines can send messages much faster because the machines themselves can be used to decode the messages. For these, a character is represented by a pattern of pulses, each pattern being of the same length. Instead of dots and dashes, which are two different lengths of electric pulses, they use one basic time interval and in that time interval have either a pulse or a pause. Each character requires 5 basic time intervals and is represented by a sequence of pulses and pauses. We often write down a pulse as a 1 and a pause as 0, and then the pattern for B is 10011, I is 01100, L is 01001. The word BILL would be transmitted as

1001101100010010 1001

Strings of ones and zeros like this can be associated with numbers in the *binary system.* In the decimal system the number 342 means

$$3\times10^2+4\times10^1+2\times10^0$$

where 10^2 stands for 10 squared, 10^1 for 10 to the first power, that is 10, and 10^0 for 10 to the power zero, which has a value 1. In the binary system of numbers 1101 means

$$1\times2^3+1\times2^2+0\times2^1+1\times2^0$$

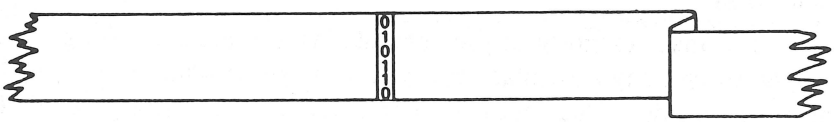
In the *decimal system* this binary number has a value $8+4+0+1=13$. We say that this number in the decimal system requires 2 *decimal digits* to represent it. In the binary system it requires 4 *binary digits.* We call a binary digit a *bit.* So the binary number representing the word BILL has 20 bits, each letter requiring 5 bits. Sometimes we take the number of bits required to represent a character as a group and call it a *byte.* Then the

word representing BILL has four bytes. In a computer we must have a way of recording these bits, and usually the memory is arranged into *words*, each capable of holding a whole number of bytes.

In some machines a single letter is represented by a byte of six bits and the word length is 6 bytes or 36 bits. Most minicomputers and microcomputers have 2 bytes in a word and 8 bits in each byte. There are many different combinations of byte length and word length in different computers. This is something the machine designer must decide.

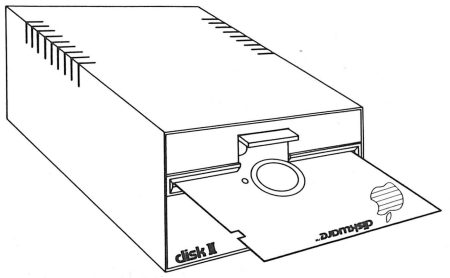
MEMORY

Most machines record letters and numbers in the binary form because it is possible to have recording devices that can record, read, and hold such information. Most recording devices involve a recording something like that on the tape of a magnetic tape recorder. There is a big difference, though, in the recording. On audio tape we have a magnetic recording that varies in intensity with the volume of the sound recorded. The frequency of the variations gives the pitch of the sound. For a computer, the recordings vary between two levels of intensity which you might think of as "on" and "off". If in a particular region there is an "on" recording it could indicate the binary digit one and if "off" the digit zero. So on a strip of magnetic tape there would be designated areas that are to hold each bit of information.



BITS RECORDED ON MAGNETIC TAPE

Binary digits can thus be recorded on reels of magnetic tape. In a similar way they can be recorded on tracks of a magnetic disk. To read or record information on a magnetic disk the recording/reading head moves to the correct track of the disk and the disk is set spinning so that the information on the track can be accessed.

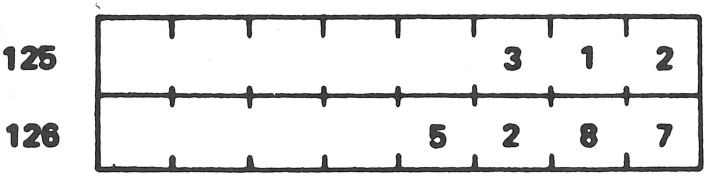


MAGNETIC DISK MEMORY

Tape reels and disks can be removed from the machine and stored if you need to keep information for long periods of time. Small disk memories are often used on microcomputers where individual disks are inserted into the reader by the user. These disks are often limp and are called *floppy disks* or *diskettes.* Diskettes are kept in paper envelopes to protect them. They are *not* removed from the inner envelope even during use. Tape reels on regular audio-tape cassettes can also be used as secondary memory. We will be talking only of diskettes in this book. The use of audio-tape cassettes results in very slow operation whenever secondary memory is required.

Both tape and disk secondary memories require the movement of objects, a reel of tape, a spinning disk and sometimes read/write heads. These mechanical devices can never give really high-speed access to information. We need memory devices with no moving parts so we can perform operations at rates of the order of a million a second. The only things that move in a really high-speed memory device are the electric signals. As you know, electric signals can move very rapidly, at nearly the speed of light. A very common form of high-speed memory uses large scale integrated circuits (LSI).

The main memory of the computer is a number of bits grouped into words; to find any particular word you must know where it is located in the array of words. You need to know its *address.* Every word (which, remember, is just a group of bits) has its own address which is a number. An address may, for example, be 125. Words that are neighbors in the array have consecutive addresses, such as 125 and 126, just like apartments in an apartment house. The addresses themselves do not have to be stored in the computer. You can tell what address a word has from its location in the array. This is not always possible for apartments because the numbering is not completely systematic.



THE CONTENTS OF WORDS IN MEMORY

Since the development of microminiaturized electronic circuits on silicon chips, high-speed memory devices holding thousands of bits can be produced using as little as a single chip.

ARITHMETIC UNIT

All computers have a part where arithmetic can take place. This is the *arithmetic unit.* When a new number is written in a memory location, the old number stored there is automatically erased, just as any old recording is erased as a new recording is made on a magnetic tape recorder. Just reading a number, like playing an audio recording, does not damage the recording no matter how often you do it. If you want to combine numbers, say add them, it is usually done in a special location in the arithmetic unit called the *accumulator.* On some machines, the size of the accumulator is the same as the size of a word in memory. Words, or rather the information stored in memory locations, can be loaded into the accumulator. In a simple machine language, the instruction

LOAD 125

would cause the number recorded in location 125 to be placed in the accumulator. Whatever was recorded in the accumulator before would be erased before the load takes place. If we want to add another number we would write

ADD 126

This would add the number stored in 126 to what was already in the accumulator and the sum of the two numbers would then be in the accumulator. This total could be recorded in the memory for later use by the instruction

STORE 127

The result of the addition would now be in location 127 but would also remain in the accumulator.

The accumulator can also be used for subtraction, multiplication, and division. In a high-level language like Basic you never need to think about

the accumulator. You merely indicate that you want numbers in two locations, say A and B, to be added and name the location, say C, where you want the answer to be stored. You write this all in one statement, namely

```
C=A+B
```

This Basic statement says: add the number stored in location A to the number stored in location B and place the result in location C. In the machine all location addresses are numbers. In Basic we give the locations names which are called *identifiers.* The interpreter changes these names to numerical locations and changes the single Basic instruction

```
C=A+B
```

to the three machine instructions.

```
LOAD  A
ADD   B
STORE C
```

CONTROL UNIT

You have just seen examples of machine language instructions. They each consist of two parts: the operation part, for example LOAD, and the address part, A. Each part can be coded as a binary number, then the whole instruction will just be a string of bits. Suppose that you have a machine with a word length of 36 bits. Then an instruction might be itself stored in such a word with, say, 18 bits for the operation part and 18 bits for the address part. With 18 bits you can represent binary numbers that go from 1 up to 2 to the power 18, which is 262,144. You can refer to any one of over a quarter of a million different memory locations.

Consecutive instructions in a machine language program are stored in consecutive locations in the memory and are to be executed one after the other. The control unit does two things. It uses a special location called the *instruction pointer* to keep track of what instruction is currently being executed. It places the instruction to be executed in a special location called the *control register.* In the control register the instruction is decoded and signals are issued to the different parts of the computer so that the operation requested is actually carried out. As each instruction is executed, the instruction pointer is increased by one to give the address of the next instruction in the program. This next instruction is then fetched from the memory, placed in the control register, and executed. This process continues, with instructions being executed sequentially unless a special instruction is encountered, which resets the instruction pointer and causes a *jump* from the normal sequence to a different part of the program.

In brief, the control unit controls the sequence of execution of instructions and determines the effect that execution has on the information stored in the memory.

Computers were originally referred to as *stored program calculators* because the instructions as well as the numbers or characters they operate on are stored in the memory. They were also referred to as *sequential machines*, because normally they followed a sequence of instructions one after another unless a jump instruction directed them to do otherwise.

We have said that the memory of a computer can be contained on a single silicon chip. This is true also of the parts of the computer that make up the central processing unit. If the CPU is all on a single chip we call the computer a *microprocessor.*

INPUT AND OUTPUT

We have spoken of putting both data and instructions in the memory of the machine and changing the data by the execution of instructions. But how do we get data or instructions into the computer, and how do we get data out of the machine after it has been operated on? That is the function of the *input and output units.* We must have instructions that cause the machine to *read* information into its memory and to *write* information out from its memory. And we must have parts of the computer, the input and output units, that respond to these instructions. Most small computers use individual input/output terminals at which information can be entered through a keyboard and output is produced as a display on a cathode ray tube (CRT) screen. Sometimes there is also a printer for obtaining *hard copy* of what can be seen on the screen. Because the user enters the program and data directly into the computer and receives the results rapidly, the system is often said to be an *interactive* system.

The keyboard of a computer is similar to that of a standard typewriter, so it helps if you can type. But hunt-and-peck methods will get you there too. In addition to the ordinary typewriter keys, there are special keys for indicating that you want the computer to take certain actions. These keys are not the same from one kind of microcomputer to another. The hard copy output units can be line-at-a-time printers or typewriters. Across a displayed page there are often positions to output 120, 132 or more characters although some printers output only 72 characters on a line. The paper is continuous but may be divided by perforations into pages, each capable of holding about 60 lines of output.

PROGRAM TRANSLATION

We have said that three machine language instructions, namely,

```
LOAD    A
ADD     B
STORE   C
```

correspond to what is written in Basic as

```
C=A+B
```

Instructions in the high-level language Basic are very much simpler to write than instructions in machine language. For one thing, you do not have to be aware of the accumulator; for another, the notation is very similar to the one used in simple mathematical expressions and should be easy for you to get used to. The Basic language is more powerful in that a single Basic instruction can correspond to many machine language instructions. We will see later that if you are working in a high-level language, the machine can detect when you make certain kinds of mistakes in your program.

In summary, high-level languages are designed to suit *you* rather than suit a computer. And in being that way they make the job of programming less difficult.

A Basic program cannot execute directly on a computer but must be interpreted for the particular computer you have. This is accomplished after the Basic program has been entered into the memory. The interpretation is performed by another program, called the interpreter, that is already stored in the computer memory. The interpreter reads your Basic program one statement at a time and produces the appropriate sequence of machine language instructions from your Basic statement. Interpretation continues provided you have not made any errors in your Basic program that the interpreter can detect. The kind of errors that are detectable are mostly in the *form* of the statements. If they are not proper or grammatical statements in the Basic language the interpreter will report an error to you on your screen. Errors in grammar are called *syntax errors.* In English you know there is an error in the sentence,

THE BOYS IS WALKING.

A machine can spot this kind of error but it cannot easily spot an error in meaning. It might never determine that the sentence,

THE HOUSE IS WALKING.

is not a meaningful sentence; it would accept it as syntactically correct.

Well, that is enough of an introduction now; let us get down to actually writing programs.

CHAPTER 2 SUMMARY

In this chapter we presented the main parts of a computer and showed how information is stored in the memory. We explained briefly how a high-level language such as Basic is interpreted by translating each statement before being executed by the computer. The following important terms were introduced.

Memory - the part of a computer that stores information, such as data or a program. Magnetic tapes and disks are called secondary memory they require mechanical motion to access information stored on them. Main memory can be immediately accessed by the computer; main memory is usually in the form of a microminiaturized circuit on a silicon chip. We say the memory is a solid state one. The computer can transfer information between secondary memory and main memory.

CPU (central processing unit) - composed of the arithmetic unit and the control unit. The arithmetic unit carries out operations such as addition and multiplication. The control unit directs other parts of the computer, including the arithmetic unit, to carry out a sequence of instructions that is in the main memory. If the CPU is contained on a single silicon chip we call it a microprocessor.

Input and output - ways of getting information into and out of a computer. An output device called a printer takes information from main memory and outputs it on paper. This is referred to as hard copy. An input/output device suitable for use by individual users is often a keyboard and a cathode ray tube display screen attached to the computer.

Screen - a cathode ray tube used to display the output from a computer.

Keyboard - the array of keys used to enter data and instructions into a computer.

Bit - a digit in the binary system of numbers, usually written as a 0 or a 1.

Byte - a sequence of bits treated as a unit, for example, the number of bits used to represent an alphabetic character.

Word - A number of bits grouped as a unit for holding information in the memory of a computer. Each word has an address which indicates its location in the array of words in memory. A word often consists of four bytes.

Coded information - before information can be entered into a computer, it must be coded in a convenient form for the computer's circuitry. The circuitry recognizes "off" and "on" which we can think of as 0 and 1.

Interpretation - before a program written in a language like Basic can be executed by a computer, it must be translated one statement at a time into a language that can be interpreted by the computer. The program as written in Basic is called the source program.

Microprocessor - a computer in which the entire CPU is contained on a single silicon chip.

Syntax error - an error in the form (or grammar) of instructions.

Semantic error - an error in the meaning of a program.

Accumulator - the part of the CPU where arithmetic operations are performed.

Control unit - the part of the CPU that determines the sequence of execution of instructions.

Chapter 3

GETTING THE COMPUTER TO CALCULATE AND OUTPUT

This is the chapter where we set the stage for programming and you meet the cast of characters in the play. We will first tell you how to give single Basic instructions to the computer which will cause it to perform calculations and display the results. Then we will show you how to prepare a list of several instructions, which is what a program is, which will not be executed until you give the computer the command to RUN the program. We say execution is *deferred.* Writing programs is what we will be doing for most of the book but to begin we will look at the *direct* or *immediate* execution of single instructions. But first we will introduce a few terms we must understand before we go very far.

CHARACTERS

You will be learning the programming language Basic a little bit at a time. Any language consists of words and the words are made up of *symbols* that we call *characters.* These characters are put together in *strings.* In English the word

ELEPHANT

is a string of characters of length eight. It contains only seven different characters, the character E being used twice. We can tell that it is a word because it has a blank in front of it and one at the end. In a way the blank is also a character, but a *special character* for separating words. We sometimes denote the blank by b when we list the output from programs in this book so that you can see how many blanks are present.

In English, we group words into sentences and we can tell the end of a sentence because of a special mark, the period. We also have a different kind of sentence that ends with a question mark, don't we? In addition to periods and question marks, we have other *punctuation marks* which serve to make sentences in the language easier to read. They also serve to remove ambiguity in a sentence. There is some doubt about the meaning

of the sentence,

THE STUDENT CLAIMS THE TEACHER UNDERSTANDS.

The doubt is removed if it is written with commas, as,

THE STUDENT, CLAIMS THE TEACHER, UNDERSTANDS.

It is important that statements in a programming language be *unambiguous*, so punctuation is used. Instead of a sentence, the basic unit in the main part of a program is a *statement.* Statements are input on separate lines or on the same line separated by colons. This serves to separate them just as periods separate sentences in English. The comma is used to separate items in any list of similar items, and parentheses are used to enclose things that belong together.

We will have words in Basic that are made up of letters of the alphabet and might also have digits in them. When we are writing a Basic program we want it to be understandable. The Basic language contains keywords that are in English like PRINT and IF. Also we can include words we ourselves make up. We choose words like NAME, INCOME, TAX, INVOICE, SUM, and so on. You are not allowed to use words that do not have at least have one letter at the beginning. If the Basic interpreter sees a digit at the beginning of a word it assumes that it is a *number.* For example, 317 is taken as a number. This means that words like 3RDPAGE are illegal and will not be accepted by the Basic interpreter.

Before we leave characters we should perhaps list them. A *character* is a letter, or a digit, or a special character. The *letters* are

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z

In this book we will use capital letters only, in our programs.

The *digits* are

0 1 2 3 4 5 6 7 8 9

The *special characters* are

: + - * / = , . " ( ) < > ; ^
b blank
as well as # $ % & ' @ ?]

Pairs of special characters are often used as *special symbols* in the language. For example, each of the following pairs has a fixed meaning

= <> <= >=

NUMBERS

Computers can do arithmetic calculations and they can do them extremely rapidly. When you learned arithmetic you first learned to handle numbers that are whole numbers, or integers. You learned that 5+6=11 and 2x3=6. In Basic numbers like 2, 3, 512, 809, and 46281 are called *integer constants.* Any string of digits is an integer constant. You will remember that we will be storing numbers in the computer and representing them as a string of bits in some coded representation. The largest integer we can represent will be limited by the length of the string of bits that are used to represent the number in our computer. This varies from one computer to another and different Basic interpreters have different maximum lengths for the digit strings that represent numbers. In the Applesoft Basic numbers are stored and can be printed up to nine digits.

If you have integers requiring longer digit strings, for instance the population of the world, they will be input or output in the form of numbers which is the *real* form (sometimes called the *exponent* or *floating point* form).

If you have a large number like

1, 635,642,421

you can write it as

1.63564242×10^9

Perhaps you recognize this as what is called scientific notation. In Basic the form of a *constant* such as our example is

1.63564242E9

The first part is called the *fraction part* or *mantissa*, the second part the *exponent.* The exponent part is written using the letter E followed by the power of 10 that is to multiply the fraction part. Maybe you learned this notation before in a science course where very big numbers, like the mass of the moon, often occur. Reals are limited in Applesoft Basic to being between -1E38 and 1E38.

Another notation is also used for numbers that are not integers. These are either *fractions* or *mixed numbers.* We write either of these in decimal notation where a point called the decimal point separates the integer from the fraction part. Examples of fractions are

.5 .0075 .0000023

Mixed numbers are

5.27 889.6 6.0216

This is called *fixed point* notation.

In the computer, all numbers are stored as real numbers unless we specifically state they they are to be integers. Input of numbers can be in any of the three forms: integer, exponent, or mixed. In output, if the number is an integer it is printed as such. If it is real, it is output in the mixed form if possible, otherwise it is output with an exponent. Leading zeros before a decimal point are not output, nor are the trailing zeros of a real number.

CHARACTER STRINGS

We have said that computers can handle both numbers and strings of characters.

A character string can consist of any of the characters that we have specified: letters, digits and special characters. Very often, when displaying the results of a computer calculation, we want the results labelled. What we want is to output a string of characters on the screen. In the statement that specifies what we want the computer to output, we include the actual string that we want displayed, enclosed in quotation marks. These strings enclosed in quotation marks are called *literals* or *character string constants.*

Examples of literals are

"BILL JONES", "BALANCE IN ACCOUNT", "X="

EXPRESSIONS

One of the important concepts we have in Basic is that of an *expression.* The way that we explain what a word like expression means is basically to give examples and then generalize these examples.

First of all 32, 5, 2.317E10, 6.1 and 58.1 are all expressions. So the general statement is that integer constants and real constants are expressions. So are literals like

"THIS IS AN EXPRESSION"

Any expression may be enclosed in parentheses and still be an expression. For example, (32) and (6.1) are also expressions. The expressions that are integer or real constants can be combined into compound expressions using the signs of arithmetic for adding, subtracting, multiplying, and dividing. These expressions are called *arithmetic expressions.* We use the standard

signs for adding and subtracting, namely the plus and minus. For multiplication we use the asterisk (*) because there is no times sign. For division we use the slant or slash symbol (/). For exponentiation we use the caret ^. Examples of arithmetic expressions are

2+3, 5.2*7.8, 6/2, 10-15, 2^3

Here is a very complicated arithmetic expression

2*5+8-3*5/2+6

In evaluating this you have to know what to do first because you really can only add, subtract, multiply, or divide numbers two at a time. The rule is to do the multiplications and divisions first, then the additions and subtractions. Also you start at the left-hand side of the expression and work to the right. We are using here *rules of precedence*: that the operations multiply and divide have precedence over add and subtract. Parentheses can be used to guide the sequence of evaluation. For example, you write 3*(5+8) instead of 3*5+8 if you want the addition to take place before the multiplication. Expressions in parentheses take precedence. Exponentiation has precedence over multiplication and division.

EXAMPLES OF ARITHMETIC EXPRESSIONS

The following examples illustrate the rules for performing arithmetic in the Basic programming language.

72+16	Value is 88.
8*5+7	Value is 47. Note that * means multiply.
2+10*4	Value is 42. Note that multiplication is done before addition.
(2+10)*4	Value is 48. The parentheses cause the addition to be done before the multiplication.
1/3	This division will produce the real value .333333333,

(9.83E0+16.82E0)/2.935E0

This expression is equivalent to the following

$$\frac{9.83+16.82}{2.935}$$

Real numbers may be written in the exponent form or as mixed numbers. The parentheses were used so the division would apply to the sum of 9.83E0 and 16.82E0 (and not just to 16.82E0).

5^3 Value is 125.

OUTPUT

Our main purpose in this chapter is to introduce you to Basic and to get you to write your first Basic instructions. The instruction which we will use first is the PRINT instruction. Then you can see that some action is taking place.

The statement that we will begin with is like this

```
PRINT 3, 5.1, "BILL"
```

Output produced by the PRINT statement is placed left-justified in three successive fields across the output line. The line is 40 characters wide. The first two fields are 16 characters wide. The third field is 8 characters wide. The items that are separated by commas after the PRINT are placed one to a field going from left to right and then a new line is started. Literals are displayed, without the quotation marks. Real numbers are displayed as mixed numbers if possible. If they are too large or too small they are displayed in exponent form. The exponent is represented by a capital E followed by the sign of the exponent, then the integer which is the power of 10. The output for the PRINT statement we showed would be

3bbbbbbbbbbbbbb5.1bbbbbbbbbbbbbBILL

We have written b for blank so you can count the spaces.

Now it is time to test this instruction for yourself. Place the disk operating system (DOS) diskette in the slot of the Apple disk drive then turn on the power to the computer and to the display screen. After some whirring noise from the disk drive the prompt character] and the blinking cursor will appear on the screen at the upper left corner. We say we have *booted* the disk operating system. You can now enter the Basic instruction we just gave. Be sure to press the return key after entering the instruction. Try another, say

```
PRINT 4,3,2,1, "BLAST OFF"
```

What happened this time? A variation in output format may be obtained by separating the items to be output by semicolons instead of commas. This causes the output all to be run together. For example

```
PRINT 3; 5.1; "BILL"
```

will result in

35.1BILL

A blank line can be left by using

```
PRINT
```

If you want to clear the screen and start fresh in the upper left-hand corner, enter the Basic instruction HOME (followed by return, of course).

Expressions other than integer and real constants and literals may also be placed in a PRINT statement. The statement

```
PRINT 2+3, 4/2
```

will result in a 5 being displayed in the first field and 2 in the second. An interesting statement might be

```
PRINT "2+3=";2+3
```

It would output

2+3=5

Note that the quotes around the literal are not displayed and that the two output items are run together because of the semicolon. You can now use the PRINT instruction to test the results from all the arithmetic expressions in the previous two sections. Try them.

You can control the format in a PRINT instruction by using the TAB(*n*) control. For example, the instruction

```
PRINT TAB(10); "HELLO"
```

outputs the word HELLO with the H starting in column 10. Try this instruction

```
PRINT TAB(1);1;TAB(3);2;TAB(5);3
```

Note that the items are separated by semicolons.

If we use ;SPC(*n*); in a PRINT instruction between values to be output, *n* spaces will be left between them. For example

```
PRINT 1;SPC(2);2;SPC(3);3
```

will result in the output

1bb2bbb3

THE PROGRAM

Now that you know two kinds of statement that will give some immediate action, PRINT and HOME, you must learn what is necessary to make a complete program which is a list of instructions that are not executed until you give the command to RUN the program. We say the execution is deferred until the entire program has been entered. In a program each line is labelled with a number on the left-hand side. This label indicates the position that the line is to have in the list. The numbers need not be consecutive but the order of the *line numbers* (or *labels*) gives the sequence of the instructions in the program. We often number by 10's in case we later have to insert other lines in the program. It is then not necessary to renumber the ones we have.

In order to store programs on the diskette for future use, the program must have a name, so we will adopt a practice of having the first line of each program give its name. To do this we make use of the Basic statement REM which is short for *remark*. We will write for our program

```
010     REM PROGRAM OPUS1
```

The name OPUS1 is one we made up to describe the very first complete program. (OPUS is Latin for "work".) You must make up an identifier you like yourself. We will number the first line 010 since we will be numbering lines by 10s. After the REM which is a *keyword* in Basic we put the word PROGRAM, then the name we have chosen for our program.

Now comes the big moment for a complete program.

```
010     REM PROGRAM OPUS1
020     PRINT " 2+3=";2+3
030     END
```

There it is, our opus number one, a complete Basic program. The body of the program starts after the remark giving the name of the program and finishes with the keyword END. In between is a list of statements one to a line with line numbers to the left. Our list consists of only one statement, the PRINT statement. Now we can try our program. First give the system command

```
NEW
```

This insures that the memory is clear. Now you can enter your program. Type it carefully into the computer. If you make a mistake in any line you can simply retype the line. When you have finished, the prompt symbol and cursor will be beneath your program. Now give the operating system command to execute the program: Type RUN followed by return. The computer should output

```
2+3=5
```

ANOTHER PROGRAM

This program illustrates the use of several PRINT statements.

```
010   REM PROGRAM ZIGZAG
020   PRINT "Z     G";TAB(7);"Z     G"
030   PRINT " I   A ";TAB(8);"I   A"
040   PRINT "  GZ  ";TAB(9);"GZ"
050   END
```

Try entering this program; be sure you give the command NEW so that the old program will be erased from the memory before you begin. The program causes the following pattern to be displayed.

```
Z     GZ     G
 I   A   I   A
  GZ     GZ
```

As you can see, the top line of the pattern is displayed by the first PRINT statement. This statement causes its first literal, " Z G", to be displayed in the first field of the output line and its second literal, again "Z G", to be displayed starting at position 7 of the output line. This is because we have used the TAB control. The next two statements cause the output of the second and third lines of the pattern.

Here is another example to try:

```
010   REM PROGRAM TEST
020   PRINT "2+3=";
030   PRINT 5,
040   PRINT 7
050   END
```

Notice that one of the PRINT statements ends with a semicolon and one ends with a comma. Run this program to see what effect this has on the output. You will see that output continues on the same line.

PRINTED OUTPUT

If you have a printer attached to your computer you can direct the output to the printer instead of the screen. This is done by giving the system command

```
PR# 1
```

(where 1 is the slot of the machine in which the printer is attached) before you give the RUN command. To return the output to the screen you must give the command

```
PR# 0
```

Usually a printer has more character positions in a line than the screen and lines that are split into two on the screen may appear as a single line on the printer.

These commands may be used as statements in a Basic program so that, within a program, output can be directed to either the printer or the screen. For example, the program

```
010    REM PROGRAM ONOFF
020    REM OUTPUTS ON SCREEN AND PRINTER
030    PRINT "MY NEXT WORD IS"
040    PR# 1
050    PRINT "HERE WE ARE ON THE PRINTER"
060    PR# 0
070    PRINT "MY LAST"
080    END
```

will output this display on the screen

```
MY NEXT WORD IS
MY LAST
```

and the line

```
HERE WE ARE ON THE PRINTER
```

will appear on the printer. (Be sure to turn the power on to the printer before you run this program.)

CHAPTER 3 SUMMARY

In this chapter, we explained how to write single Basic instructions for immediate e xecution and to write very simple computer programs for deferred execution. The following important terms were presented.

Character - is a letter (ABC...Z), digit (012...9) or special character +*/.;:, etc.

Integer constant - is an integer (whole number) such as 78 and 2931. There may be a minus sign in front of the integer.

Real constant - is a number such as 3.14159E0 (equal to 3.14159×10^0 or simply 3.14159). A real constant in the exponent form consists of a fraction (3.14159) and an exponent part (E0).

Literal (or character string constant) - is a sequence of characters enclosed in quotes, such as "WHY NOT?".

Arithmetic expression - composed of either a single number or a collection of numbers combined using addition, subtraction, multiplication, division, and exponentiation (+, -, *, / ^). Parentheses may enclose parts of the expression.

Rules of precedence - specify the order for applying +,-, * and / to compute the value of an arithmetic expression. Parenthesized expressions are evaluated first. Proceeding from left to right, ^ is applied first, * and / are applied second and then + and -.

PRINT - means output either to the screen or the printer. The PRINT statement prepares expressions (literals or arithmetic expressions) for output. A line is not actually completed if the last output item in a PRINT statement is followed by a comma or semicolon.

PRINT - integers, real values, and string literals such as -24, 2.7, and "HELLO" are displayed using a statement of the form

PRINT list of expressions separated by commas

The values are displayed on the screen. A real number is displayed in the mixed number form if possible. Insignificant zeros are not output. Up to 9 significant digits are displayed. For example,

```
PRINT 1/3
```

results in the output .333333333.

Field - PRINT causes values to be displayed in three fields across a line. Numbers and literals are left-justified in the fields which are 16, 16, and 8 characters wide. The output line is 40 characters on many screens. Sometimes the third field appears on the next line.

Formatted output - using the TAB or SPC features in the PRINT statement so that the output line is well spaced. For example

```
PRINT 2;TAB(3);3;TAB(6);4;TAB(9);5
```

produces the output

```
2b3bb4bb5
```

Notice that the number in parentheses after the keyword TAB gives the column where the first character of the next output item will be placed.

A fixed number *n* of spaces can be put between output items by having ;SPC(*n*); between them in the PRINT statement.

Output - display on a cathode ray tube screen, which the computer does at your request. The PRINT statement produces output from the computer.

Printed output - can be obtained by redirecting the output produced by the PRINT statement in a program from the screen to a printer. This is done by giving the system command

```
PR# 1
```

where 1 is the number of the slot where the printer is attached. To restore the output to the screen you must give the command

```
PR# 0
```

These commands can also be used as statements in your program.

CHAPTER 3 EXERCISES

1. What will the following program cause the computer to output?

```
010   REM PROGRAM LETTER
020   PRINT " *    *"
030   PRINT " **  **"
040   PRINT " * * *"
050   PRINT " *    *"
060   END
```

Can you rearrange the lines in this program to output a different letter?

2. What do the following cause the computer to output?

(a) PRINT 2," PLUS ",3," IS ",2+3
(b) PRINT "234+198+362=",234+198+362
(c) PRINT "2 FORMULAS:",2+3*5,(2+3)*5
(d) PRINT "SUBTRACTION",20-10-5,20-(10-5)

3. Write statements to calculate and output the following:

   (a) The sum of 52181 and 10032.
   (b) 9213 take away 7918.
   (c) The sum of 9213, 487, 921, 2013 and 514.
   (d) The product of 21 times the sum of 816, 5 and 203.
   (c) 343 plus 916 all multiplied by 82.
   (f) 3.14159 (π) times 8.94 divided by 2.
   (g) 3.14159 times the square of 8.94 (Note: X^2 can be written as X*X or X^2).

Chapter 4

EDITING AND SAVING PROGRAMS

So far we have looked at the immediate execution of Basic statements and the preparation of simple programs for deferred execution. To achieve immediate execution we simply type a single Basic statement (without a line number) followed by return. At present we really have only two statement types to try: PRINT and HOME. As soon as the return is pressed the statement is executed.

To enter a program we type a series of statements that have line numbers. The series is terminated by the END statement. When the system command RUN is given the series of statements is executed in order of ascending line numbers.

In order to start entering a new program and to get rid of the previous program in the memory we preceded our typing of the new program by the system command NEW.

INITIALIZING YOUR OWN DISKETTE

When you want to save a program that you have written, it must be recorded on a diskette. The diskette that we used to begin our session at the computer was the one containing the Disk Operating System (DOS). If you want to save your own programs you should have a separate diskette of your own. We will now tell you how to prepare a diskette of your own. After you have this diskette you will begin your sessions at the computer by placing your own diskette in the disk drive and turning on the power. When you do this, the computer will automatically execute the initial program stored on your diskette. So we will now write a program that will be suitable as the one to be executed. It should give you a message to confirm that you put the right diskette in the drive. Here is such an identification program:

```
010     REM PROGRAM ID
020     HOME
030     PRINT "DISKETTE 001 OF PAT HUME"
040     PRINT "SYSTEM IS READY"
050     END
```

Type in this program (using your own name, of course) just as you did the other programs we tried in the previous chapter. (Be sure to give the NEW command before you start.) Now give the RUN command; if the display says:

```
DISKETTE 001 OF your name
SYSTEM IS READY
```

then you can proceed to the next step. Be sure to follow instructions carefully.

Remove the DOS diskette from the drive and put in a new diskette (or one that is no longer needed). Shut the door of the drive. Now give the command

```
INIT ID
```

This will cause the disk drive to whir and the diskette will be initialized for use and have the ID program as its first program. The display created by the ID program will appear on the screen.

You can now check to see that this all works. Take your new diskette out of the drive and turn off the computer. Write on the diskette envelope the identification, for example, 001 PAT HUME, so that you can identify it yourself. Place the diskette back in the drive and turn the power back on. If it is working the identification message will appear on the screen. Now you are ready to use the diskette to save your programs.

SAVING A PROGRAM

To save a program: type it in as usual, then, after you are satisfied that it works, by giving the RUN command, give the command

```
SAVE name of program
```

This name should be the one following the word PROGRAM in the first line of the program. Names can be up to 30 characters long and begin with a letter. Names can contain any characters including blanks but with the exception of the comma.

Here is a program to try. Enter it right away.

```
NEW

010     REM PROGRAM HI
020     PRINT "HAVE A GOOD DAY"
030     GOTO 020
040     END
```

When you type RUN, to test the program, you will find that it gives results that are a bit repetitious. As the screen fills, the lines move upward, or *scroll*, so that the next line can be displayed. You can interrupt the program by typing CTRL-C (press the *control key* and while you are holding it down press the C key). If you want it to continue type CONT. You may find that the scrolling is so fast that you can hardly read anything. You can slow down the rate at which characters are sent to the screen by giving the command

```
SPEED = 130
```

This sets the display to half speed. To restore it to normal speed we would give the command

```
SPEED = 255
```

This is top speed. Try slowing down the display for this program. What is like at a speed of 25?

We do not want to go on with all this well-wishing but we do want to save the program. Give the command

```
SAVE HI
```

To see that you have saved the program give the command

```
CATALOG
```

You will see displayed a list of programs that you have saved. It will say:

```
A   002   ID
A   002   HI
```

The A is the sign that it is an Applesoft Basic program. You now have two programs in your disk file. To run a program in the file you type RUN followed by the name of the program. If you type

```
RUN ID
```

You will get the identification display. To see what program is currently in the high-speed memory give the command

```
LIST
```

You should see the ID program listed. If you want only one line of the program in memory listed, type:

LIST number of line

If you want a range of lines listed, type:

LIST first line number - last line number

If the first line number is omitted here it is assumed to be the first line of the program. If the last line number is omitted it is assumed to be the last line of the program. For example,

LIST 50 -

will produce a listing from line number 50 to the end of the program.

You will notice that the listing of a program is not spaced in the same way you have typed it. Also a line number such as 010 will appear in the listing as 10. We used the form with the zero in this book to get a better alignment of the type. Most lines in our programs have three-digit line numbers.

A long listing that more than fills the screen can be stopped so you can look at it by typing CTRL-C; typing CONT permits it to continue.

Sometimes you do not want to run a program that you have stored on the diskette but want to bring it into memory so that you can make changes in it. This can be accomplished by typing:

LOAD name of program

EDITING A PROGRAM

We have mentioned several ways of editing a program as it is being typed into the memory. If you type a wrong character press the left-arrow key (←). This moves the cursor left one character position and erases the character. (The character in the display on the screen is not erased but just ignore that.) You can move left over as many characters as you want to erase. Now just type the correct characters.

To correct a line that has already been finished, that is, by typing return, the simplest thing is to retype the whole line; the new line replaces the old line with the same line number.

To delete a line, type its line number followed by return. Deleting a sequence of lines is done by typing the command

DEL first line to be deleted, last line to be deleted

All lines with numbers between these lines are deleted.

If you are typing a line and you want to abort the line, just type CTRL-X followed by return. This tells the computer to forget it.

So far we have been speaking about replacing whole lines. It is possible to make changes within lines but this is slightly more difficult. What we must do is to list the line to be corrected using LIST, then move the cursor to the place where the correction is to occur. To move the cursor you must press the escape key (ESC) followed by one of the letters A, B, C, or D. (Escape D is unlike a CTRL-D in that here you press the escape key, release it, and the press D, whereas for CTRL-D you hold the control key down while you press D.) Here are the four possibilities:

escape A moves cursor right
escape B moves cursor left
escape C moves cursor down
escape D moves cursor up

Unlike the left and right arrow keys the escape A or B do not affect any characters as they move over them.

Let us suppose that line 030 in our program is in error. List the line by giving the command LIST 030. For example, here is a display you might get:

```
]LIST 030

30 PRINT "SAY HELO

]□
```

The cursor is shown as □ here. We want to put another L in HELO and add quotation marks after the O. First use the cursor moving commands to place the cursor over the first character in the line, namely 3. Now use the right-arrow key to move the cursor over to the O. The use of the right arrow copies all the characters of the line over which it has completely passed. This means that the line up to the L has been copied. (The right arrow has not yet passed over the O.)

A fast way to get a series of right arrows is to hold its key down along with the *repeat* (REPT) key. If you go too far, the left arrow lets you back up with no difficulty. To insert the L before the O move the cursor up one line by escape D so that it is above the O and type the missing L. Now move the cursor back down to line 30 by escape C. It will now be positioned just after the O. Now move it back over the O, using escape B. (Do not use the left arrow or the letter O will be erased.) Now press the right-arrow key to copy the O into the line, then type the missing quotation mark followed by return.

You can see whether the corrections have been made properly by giving the command

LIST 30

You should now see

30 PRINT "SAY HELLO"

You can make changes to any line listed on the screen; you do not have to list it again. Do not copy the prompt symbol. Replacement of characters is easy; so is deletion of characters. To delete characters: position the cursor at the first of the line then press the right arrow until you reach the first character to be deleted, press escape A over each character to be deleted, then resume pressing the right arrow up to the end of the line.

The cursor can be moved vertically to any line on the screen by the command

VTAB number of line

The lines are numbered from 1 to 24 with line 1 at the top, line 24 at the bottom. A command of the form

HTAB character position on line

will move the cursor horizontally to the stated position on the line. There are 40 character positions; position 1 is on the left, position 40 on the right of the line.

You may find all this cursor movement more trouble than it is worth. But a little practice will make it clear. Try editing a few lines yourself.

COMMANDS THAT CHANGE FILES

We have seen how to save programs in the file. We must also learn how to get rid of programs from the file. To do this give the command

DELETE name of program

To change the name of a program in the file give the command

RENAME old name, new name

You can prevent accidental losses of programs that you really want by locking them. When you give the command

LOCK name of program

you will not be able to delete, rename, or save another program by that name unless you give the command

UNLOCK name of program

A program that is locked has an asterisk beside its type designation in the catalogue.

COPYING DISKETTES

To copy a diskette when you have two disk drives is described in the disk operating system manual (DOS 3.2). We will not repeat it here.

If you have only one disk drive you must copy programs, one at a time, from the original diskette to the copied diskette. You can copy as many programs as you want and even copy different programs from different original diskettes.

First you must initialize the diskette that you are going to copy onto. This is done in the same way as we described in this chapter. You should change the identification program to suit your new diskette unless it is just to be a duplicate of the original for back up purposes.

When initialization has been done, place the original diskette in the drive and LOAD the first program to be copied into the memory. Now remove the original diskette from the drive and insert the diskette on which the copy is to be recorded. Now type SAVE giving the name of the program.

This process is repeated for each program that you want to copy. A bit boring perhaps!

CHAPTER 4 SUMMARY

In this chapter we explained how to save programs in a disk file, how to display a calalogue of what is in the file, how to load a program stored on the disk into the memory, how to run a program from the disk, and how to edit a program. To prepare a disk for storing programs we saw how to initialize it with an identification program.

To perform all these operations we gave operating system commands. The following terms were presented.

Start up - the procedure you must go through to begin a session at the computer. Before you have your own diskette initialized, start up consists of placing the diskette containing the Disk Operating System (DOS) in the disk drive and turning on the power to the computer. This is sometimes called booting the system. When you have your own diskette initialized you place it in the drive instead of the DOS diskette and turn on the power.

Initialization of diskette - to initialize a diskette for the storage of programs first prepare an identification program (ID) using the DOS diskette in the drive. With the ID program in the memory take the DOS diskette out of the drive and insert a fresh diskette, or one you do not need anymore. Then give the command

INIT ID

This causes the diskette in the drive to be initialized.

INIT - the command given to initialize a diskette for the storage of programs. The form of the command is

INIT name of identification program

When the system is booted using this diskette the named program is executed immediately.

Identification of diskette - the program that is automatically executed when the system is booted using the diskette. An identification should also be written on the envelope that contains the diskette.

Saving a program - when you want to record a program on your diskette for future use give the command

SAVE name of program

Any program of the same name already on the disk is erased by a SAVE command.

Running a saved program - a program stored on a disk is run by giving the command

RUN name of program

RUN - a program that is already in memory may be run using this command. You can, if you want, start running a program in memory at a line other than its first line by giving the command

RUN line number

Loading a program from disk into memory - give the command

LOAD name of program

Any program already in memory is erased by a LOAD. Execution of the program in memory is obtained by the command RUN.

Stopping program execution - execution is stopped by typing CTRL-C.

Execution is resumed by typing CONT for continue.

SPEED=*n* - the command that sets the speed at which characters are displayed on the screen. Speed *n* can range from 0 to 255.

Listing a program - lines of the program in memory can be displayed by a command of the form

LIST first line to be displayed - last line to be displayed

The display includes all lines whose numbers lie from the first to the last inclusive. If the first line number is omitted it is assumed to be the first line of the program; if the last line number is omitted it is assumed to be the last line of the program. To list all lines use LIST alone. To list one line use

LIST number of line to be displayed

Listing can be stopped by CTRL-C and restarted by CONT.

LOCK - the command that protects a program in the disk file from being deleted, renamed, or overwritten. The command has the form:

LOCK name or program

Displaying a catalogue of programs - the names of all the programs that you have saved can be listed by the command

CATALOG

Each entry for a program has an A in front of it, the A indicating that it is an Applesoft Basic program; a locked program has an asterisk in front of the A to indicate that it cannot be changed or deleted. The number preceding the name of the program gives the number of storage blocks for the program. You cannot delete, rename or save a program under that name unless you give the command

UNLOCK name of program

Deletion of lines from program in memory - achieved by giving the command

DEL first line to be deleted, last line to be deleted

Cursor movement - is achieved, without affecting any characters, by

pressing the escape key then, after releasing it, pressing a letter. A moves right; B moves left; C moves down; D moves up.

VTAB *n* - is the command that moves the cursor vertically to line *n* of the screen. There are 24 lines on the screen.

HTAB *n* - is the command that moves the cursor horizontally in the line to the *n*th character position. There are 40 character positions in a line.

Deletion of characters in line - move cursor to first character in line, then, using the right-arrow key, space over to the first character to be deleted. This copies the line up to the deletion. Now press escape A over each character to be deleted, then resume right arrow over rest of line.

Insertion of characters in line - position cursor on first character of line, press right-arrow character until cursor sits on top of character that is to follow the insertion, press escape D to move up one line, type insertion, press escape C to move back down to line then press escape B until the cursor is back where the insertion started. Finish by moving to end of line by right-arrow key.

Changing of characters in line - move cursor by right-arrow key from first of line to point where change is to occur, type new characters over old, then use right-arrow key to end of line.

Repetition - by pressing the repeat key (REPT) down along with another key, such as right arrow, repetition occurs.

CHAPTER 4 EXERCISES

1. If you have not already done so, initialize your own diskette with an identification program. Enter the program OPUS1 shown in Chapter 3. Run it to see that all is well. Now save it on the diskette. Type NEW to erase the memory. Load OPUS1 back into the memory and edit it so that it contains three more PRINT instructions in lines 25, 26, and 27. You invent these yourself. Change the name to OPUS2 in line 10. Now run OPUS2 to see that it works. Try the command.

   RUN 26

What happens? Save OPUS2 on the diskette. Look at the catalogue to see that it is there. Try OPUS1 again by typing

RUN OPUS1

Rename OPUS2 to be OPUS3. Try the command

RUN OPUS2

What happens? Look at the catalogue.

2. Write a program to list your five favorite TV stars in order. Save it. Now edit it so that the order is changed.

3. Write a program to produce 4 lines of output on the screen. These lines are to be spaced 5 lines apart and line 1 is to have the words "line 1" on the left, line 6 is to have "line 6" starting in column 10, line 11 is to have "line 11" starting in column 20, and line 16 is to have "line 16" starting in column 30. Try other fancy formatting.

4. Write a program to display your initials in big letters using asterisks.

Chapter 5

VARIABLES, CONSTANTS, AND ASSIGNMENTS

In this chapter you will learn how to write programs to read numerical information into the computer, to perform arithmetic calculations on the numbers you read in, and to output the answers. You will learn, as well, how to make your programs understandable to others (as well as to yourself) by careful choice of words that you can make up and by remarks that you can add to your program. The principal new concept to learn is the idea of a *variable.*

VARIABLES

We have said that a computer has a memory and that in the memory there are locations where information can be stored. Each location has its own unique address. In a high-level language like Basic we do not ever refer to an actual machine address. Instead we use a name to identify a particular location. It is like referring to a house by the name of the owner rather than by its street address. We use the word *variable* to stand for the memory location. It is named by an *identifier.*

The identifier for a variable must begin with a letter and contain no blanks or special characters. If you think of the variable as the memory location and its name as the identifier then you will realize that the *value* of the variable will be the actual information that is stored in the memory location. Locations are arranged to hold only one type of information or data. We speak of the *data type* of a variable. A variable may hold numbers, in which case we say it is an *numerical variable.* It could also be *character variable* and hold alphabetic information. If a variable is a numerical variable its value can be any number up to a certain maximum size. The value may be changed from time to time in the program but its *type* can never change; it is always a numerical variable.

Examples of numerical variable identifiers are

ACCTNO, TAX, SUM, MARK

They are similar to the identifiers we used to name a program.

It is *very* important to choose identifiers that relate to the kind of information that is stored in the corresponding locations. Well-chosen identifiers make a program easier to understand. Although identifiers may be of any length, only the first 2 characters are used by the APPLESOFT II compiler. The first character of the identifier must be a letter of the alphabet; the second character can be a letter or a digit. This means that there are 936 different possible variable names. (This is 26x(26+10).) No two different variables in the same program should have the same first two characters or confusion will result. Also care must be taken to avoid choosing identifiers that are keywords (or contain keywords) in the Basic language. A list of keywords is found in Appendix 3. These are sometimes called *reserved words.* For example, the variable identifier MORE contains the Basic keyword OR and at execution time the computer will report an error with the message

?SYNTAX ERROR

You will have to choose a different identifier.

A number of Basic keywords are tempting to use as variable identifiers, such as, DATA, SAVE, LIST, or TEXT but the little keywords are more insidious. It is easy to pick identifiers that contain small keywords such as AT, OR, TO, COS, DEL, LEN, GR, POP, POS, END, VAL, FOR, or TAB. We had trouble for the book when we picked: MONTH, WORD, VALUE, GRADE, POPULATION, POSITION, LENGTH, TABLE, SCORE, and so on. It's frustrating. You can always test any identifier name to see if it contains a key word by giving the computer the immediate execution instruction

PRINT identifier name

If it does not say

?SYNTAX ERROR

the identifier is fine. All variables are initialized to zero when you type NEW or RUN so that, if you have not assigned a value to your variable, the value will be zero.

The memory locations for the 936 possible numerical variables are predetermined for all Basic programs. This means that the interpreter can substitute the actual machine address whenever it encounters a variable in the program.

In Basic, variables can be restricted to being integers if names end in %. Since numerical variables can be used for either real or integer values, we will not use this feature of Basic.

ASSIGNMENT STATEMENTS

In this chapter you will be learning two new types of Basic statements that cause things to happen as the program is executed. We say that they are *executable statements.* The PRINT statement is an executable statement; it causes output to take place. One of the two new executable types we will have is the statement that *reads* input, the INPUT statement, but first we will look at the *assignment statement.*

There are no keywords in an assignment statement but it has a very definite form. It looks like an equation but it is not one. The form is

identifier = expression

There is an equal sign and on the left of this is a single word, a variable identifier. On the right hand of the equal sign there is an expression. We have looked at expressions that contained numerical constants; now expressions can also contain numerical variable identifiers. We have had expressions like

5+10/3 (8+9)*7

but now we can have expressions like

SUM+1 SUM/100 SUM-MARK

We will not use variable identifiers in the expression of an assignment statement to begin with but instead use a simple expression, an integer constant. For example,

AGE=5

is an assignment statement. It causes the number 5 to be stored in the memory location called AGE. Remember that only the first two characters count so this is the variable with identifier AG. If we then gave the instruction

PRINT AGE

the number 5 would appear in the first field.

So far the expression on the right-hand side of the assignment has just been a simple constant, but we can have more complicated expressions:

AGE=1984-1966

Here we are subtracting the year of birth, 1966, from the year 1984 to get the age in 1984. This instruction would assign the value 18 to the variable AGE. We could get the same result as follows

```
BIRTHYEAR=1966
THISYEAR=1984
AGE=THISYEAR - BIRTHYEAR
```

Here we have two additional variables BIRTHYEAR and THISYEAR which are given values in assignment statements and then used in an expression on the right-hand side of another assignment statement. We could have another statement

```
NEXTAGE=AGE+1
```

which would give the age the following year to the variable NEXTAGE. A variable may be assigned values over and over during a program. For example, we might have

```
SUM=2+3
PRINT SUM
SUM=3+4
PRINT SUM
```

and so on. Now we come to perhaps the most confusing type of assignment statement. Suppose in a program you were making calculations year by year and needed to keep a variable AGE that held the value of the current age for the calculation. We might change the value at the end of the year by the assignment:

```
AGE=AGE+1
```

Now you can see that the assignment statement is certainly *not* an equation, or this would be nonsense. What happens when this statement is executed is that the value stored in the variable AGE is added to the integer 1 and the result of the addition stored back in the same location.

In machine language, if the memory location of AGE is 336 and if there is a constant 1 stored in location 512, then the Basic assignment statement

```
AGE=AGE+1
```

could be translated as

```
LOAD  336
ADD   512
STORE 336
```

An alternative form of the assignment statement can be used namely:

```
LET AGE = AGE + 1
```

This form uses the keyword LET ahead of what we have been using. We will *not* use this form because most other high-level languages do not require a keyword in front of an assignment. In Basic it is optional and we will choose not to use it.

TRACING EXECUTION

We have seen that variables are associated with locations in the memory of the computer. We can assign values to variables and, during a program, we can change the values as often as we want. The values can *vary* and that is why the locations are called variables. The location stays the same but the value can change.

Sometimes it is helpful, when getting used to writing programs, to keep track of values stored in the memory locations corresponding to each variable. Some statements change a value; others do not. We call this *tracing the execution* of instructions.

We do not need to know the numerical, or machine address of the locations. As far as we are concerned the identifier is the address of the variable. For example, if before execution of

```
AGE=AGE+1
```

the value stored in the variable AGE was 13 then, after execution, the value stored in the variable AGE would be 14.

We will trace now a slightly more complicated program by writing the values of all the variables involved after each instruction is executed. Here we will use some meaningless names like X,Y, and Z because the program has no particular meaning. We just want to learn to trace execution. We will write the tracing on the right-hand side of the page and the program on the left. The labels over the right-hand side give the names of the locations; their values are listed under the names, opposite each instruction. At the beginning of execution of a program all numerical variables are given the value zero. When the value of a particular variable has not yet been assigned its value is zero. This is the case in line 010.

```
                                X     Y     Z

010   REM PROGRAM TRACE         0     0     0
020   X=5                       5     0     0
030   Y=7                       5     7     0
040   Z=X+Y                     5     7    12
050   X=X+5                    10     7    12
060   X=Z                      12     7    12
070   Y=Z                      12    12    12
080   X=X+Y+Z                  36    12    12
090   Y=Y*Z                    36   144    12
100   X=X/Z                     3   144    12
110   PRINT X,Y,Z               3   144    12
120   END                       3   144    12
```

In line 020 X has a value namely 5 the other two variables are still zero. All is straightforward until line 050 when X appears on both sides of the assignment statement. The values shown at the right of the program are, remember, the values after execution of the statement on that line.

The output statement in line 110 includes the names of variables in the list. We have

```
PRINT X,Y,Z
```

The machine can tell the difference between variable identifiers and literals because identifiers have no quotes. There is no possible confusion between numbers and identifiers because an identifier may not begin with a digit. You can see now why Basic has this rule.

In this example we showed a division. Here the answer is an integer. This is not always the case in a division. We can force the result to be an integer if we use a Basic function called INT. The statement

```
PRINT INT(25/7)
```

will yield the result 3. The actual result of the division is

```
3.5714285
```

The effect of INT when applied to a positive number is to cut off (or truncate) the fractional part leaving only the integer part of the mixed number. Sometimes we want to round off the results of a division, say in determining costs to the nearest cent. If PRICE is the cost in cents of a 2-kilogram package of soap flakes then the cost of one kilogram to the nearest cent KGPRICE is produced by using this statement

```
KGPRICE=INT(PRICE/2+0.5)
```

Notice that the constant 0.5 is added to the result of the division before INT is applied. This is one way of rounding positive numbers. We say we take them to the nearest integer.

INPUT OF DATA

Now we will learn how to read data into the computer. We did not learn this at the same time as we learned to output data because the idea of a variable is essential to input. It is not essential to output because in an output statement we can have numbers and literals, that is, integer and real constants and constant character strings. If we use

```
INPUT X,Y,Z
```

we will read three numbers that are entered on the keyboard and store them in the three variables X, Y, and Z. The numbers must be entered, separated from each other by a comma or a return. As you type, the data is

displayed on the screen. Here is a sample program that reads numbers in and outputs their sum.

```
010     REM PROGRAM ADD
020     PRINT "INPUT VALUES FOR X AND Y"
030     INPUT X,Y
040     Z=X+Y
050     PRINT "SUM OF ";X," AND ";Y," IS ";Z
060     END
```

The appearance of a sample display for this program would be:

```
INPUT VALUES FOR X AND Y
?5, 7
SUM OF 5 AND 7 IS 12
```

The first line of the display appears as soon as you give the command RUN and a question mark appears on the second. This question mark prompts you to enter values for X and Y. The cursor is positioned after the question mark. The computer then waits for you to enter the data. On input, the first number entered, namely 5, is associated with the first variable X and stored in that location. The number 7 is stored in location Y. After you have entered the 7 you must press the return key so that the numbers can be accepted by the INPUT statement of the program and the sum computed. When this happens the last line of the display appears. When a program reaches the END in execution the display will remain on the screen. Another way to get a similar result is this

```
010     REM PROGRAM ADD
020     INPUT "INPUT VALUES FOR X AND Y ";X,Y
030     PRINT "SUM OF ";X, " AND "; Y; " IS ", X+Y
040     END
```

Here the display is:

```
INPUT VALUES FOR X AND Y 5,7
SUM OF 5 AND 7 IS 12
```

First of all the use of the variable Z is eliminated but, more important, the prompting message is incorporated right after the keyword INPUT and followed by a semicolon, then the list of variables to be input. In this case no prompting question mark is output and the input continues on the same line of the screen as the prompt.

AN ALTERNATIVE WAY TO INPUT DATA

Basic provides an entirely different way to read data into a program. In this way the data is *not* entered interactively in response to a prompt from the computer but is prepared in advance and entered along with the program. The items of data that are to be input are placed in a series of numbered lines in the program and prefaced by the keyword DATA. They are placed in the same sequence as they would be entered interactively. The statement that reads this data is not INPUT but is a READ statement. An example will show how it works.

```
010     REM PROGRAM BATCH
020     REM READS DATA PREPARED IN ADVANCE
030     READ X,Y
040     PRINT X+Y
050     READ A,B,C
060     PRINT A*X+B*Y+C
070     END
080     DATA 1 , 2 , 3
090     DATA 4 , 5
```

The output display will be

```
3
16
```

The READ instruction of line 030 sets X to 1 (the first item after the keyword DATA) Y to 2 (the second). The sum 3 is output. The READ instruction of line 050 continues reading data where the last one left off so that A is set to 3, B to 4, and C to 5. If there are too many data items they are ignored; if there are too few the error results in an error message being output. If you want to cause the reading of the same data to begin again include the statement RESTORE in your program.

We will not use this method of entering data as it is associated with batch processing of programs rather than interactive use.

REMARKS

One of the main aims of structured programming is that your programs be easily understood by yourself and by others. When you create useful programs you may store them on your diskette for future use or share them with other users. Choosing variable names that suggest what is being stored is an excellent way to make programs readable. We have shown several programs with just X,Y, and Z as variable names. This is

because these are meant to show you what happens in assignment statements and INPUT and PRINT statements and are not about real applications. It is not advisable to use such meaningless names. We want your programs to look more like English than like algebra when you are finished. This is why APPLESOFT Basic permits the use of much longer variable names. The original Basic language required all numerical variables to be identified by single letters of the alphabet.

One other thing that you can do to make a program understandable is to include remarks in English along with the program. We have been providing remarks to some of our examples in the accompanying text but you can write remarks right into the program. All remarks begin with the keyword REM. For example,

```
REM THIS IS A REMARK
```

could be placed in the program. Remarks are written on separate lines or can follow another statement provided they are preceded by a colon. From now on we will be including remarks in our examples.

We have already used the remark statement to begin a program: to state that it is a program and give its name. We used this name when we stored the program on the diskette.

AN EXAMPLE PROGRAM

We now give a program which illustrates the use of variables, assignment statements, INPUT and PRINT statements, and remarks other than the standard first line of a program. The program reads in the length, width, and height of a box (as given in inches) and then outputs the area of the base of the box (in square centimeters) and the volume of the box (in cubic centimeters).

```
010    REM PROGRAM CONVERT
020    REM INPUT BOX LENGTH, WIDTH, HEIGHT IN INCHES
030    REM CONVERT TO CENTIMETERS AND CALCULATE
040    REM THE BOX'S BASE AREA AND VOLUME.
050    CMPERINCH=2.54
060    INPUT "LENGTH=";LNGTH
070    INPUT "WIDTH=";WIDTH
080    INPUT "HEIGHT=";HEIGHT
090    LNGTH=CMPERINCH*LNGTH
100    WIDTH=CMPERINCH*WIDTH
110    HEIGHT=CMPERINCH*HEIGHT
120    AREA=LNGTH*WIDTH
```

```
130     PRINT "AREA=";AREA
140     VOLUME=HEIGHT*AREA
150     PRINT "VOLUME=";VOLUME
160     END
```

This program will display the following:

```
LENGTH=2.6
WIDTH=1.2
HEIGHT=6.92
AREA= 20.129
VOLUME= 353.803
```

where the area is in square centimeters and the volume is in cubic centimeters. The area and volume displayed depend on the three values 2.6, 1.2, and 6.92 entered from the keyboard. Try this program using different values for the size of the box. The data values 2.6, 1.2, and 6.92 could be replaced by the dimensions of a different box.

Line 010 is the remark we always include to identify the beginning of the program; it causes no action on the part of the computer. Lines 020, 030, and 040 are additional remarks intended for you, the reader of the program, and are also ignored by the computer.

Line 050 of the program is a definition of a constant. The constant identified by the name CMPERINCH is given the value 2.54. (There are 2.54 centimeters in an inch.) Constants differ from variables in that they maintain the same value throughout the program's execution. We make a practice of assigning values to all such constants at the beginning of the program. This means that the number 2.54 is explained and does not appear mysteriously in line 090 as

```
LNGTH = 2.54 * LNGTH
```

Defining constants at the beginning of a program is a good habit to get into. It makes the program more understandable. We did not use the word LENGTH as a variable name because LEN is a Basic keyword. That is why we used LNGTH. Lines 060, 070, and 080 cause prompts and the data values 2.6, 1.2, and 6.92 to be read into variables LNGTH, WIDTH, and HEIGHT. Notice how the program prompts you to enter these values by displaying LENGTH= and so on right in the INPUT statement.

Line 090 takes the value 2.6 from the LNGTH variable, multiplies it by the constant CMPERINCH and then returns the result to LNGTH. The value of LNGTH will now be the length of the box in centimeters.

Line 120 takes the values in LNGTH and WIDTH, multiplies them together, and places the result in AREA. Line 130 then outputs:

```
AREA= 20.129
```

As of line 140, the variable VOLUME has not been used. An attempt to output VOLUME in line 140 would give a value zero since all variables are initialized to zero as the program begins execution.

Line 140 computes the volume and line 150 outputs the volume. Line 160 is the end of the program and tells the computer to stop working on this program.

This job would output the same thing if we made the following change. Replace lines 060, 070, and 080 by the three assignment statements:

```
LNGTH=2.6
WIDTH=1.2
HEIGHT=6.92
```

Now you do not need to enter the size of the box.

These changes result in a program which is given the dimensions of the box by assignment statements rather than by INPUT statements. The advantage of the original program, which uses INPUT statements, is that the program will work for a new box simply by entering different values from the keyboard.

LABELLING OF OUTPUT

Just as remarks and definition of constants help to make a program more understandable, output that is properly identified by labelling is self-explanatory. What you are trying to do is to prepare a display that needs no further explanation from you when you let others see your computer output. The output data should be labelled so that there is no doubt about what the numbers are, without reading the program. Prompting words serve to label the input data.

There are two basic ways to label input or output values. If different values of the same set of variables are to be entered and displayed in columns on the screen, then a label can be placed at the top of each column. For example, the display for comparing costs of boxes of soap flakes might be

```
PRICE     WEIGHT(KG)     PRICE/KG
 ? 125,       1
                             125
 ? 200,       2
                             100
 ? 260,       3
                              87
```

Here the values in the PRICE and WEIGHT(KG) columns are input values. You can include as many blanks as you like after the comma between input values on a line so that you can line up the entries with the column headings. You must press return after you enter a number in the WEIGHT(KG) column so that the computer will know that you have finished. The output data corresponding to the two input values is displayed on the next line starting in column 20 so that they line up with the PRICE/KG column heading. This is done by putting TAB(20) followed by a semicolon just before UNITPRICE.

There is no reason to use exactly the same labels as the variable names, since the literals displayed at the top of the columns can be longer and contain blanks or keywords. They are displayed independently. The program that produces this table might be

```
010     REM PROGRAM SOAP
020     REM COMPUTE AND TABULATE PRICE PER KG
030     REM PRINT HEADINGS OF TABLE
040     PRINT "     PRICE ","WEIGHT(KG)","  PRICE/KG"

050     REM PROCESS DATA FOR FIRST BOX
060     INPUT PRICE,WEIGHT
070     UNITPRICE=INT(PRICE/WEIGHT+0.5)
080     PRINT TAB(20);UNITPRICE

090     REM PROCESS DATA FOR SECOND BOX
100     INPUT PRICE,WEIGHT
110     UNITPRICE=INT(PRICE/WEIGHT+0.5)
120     PRINT TAB(20);UNITPRICE

130     REM PROCESS DATA FOR THIRD BOX
140     INPUT PRICE,WEIGHT
150     UNITPRICE=INT(PRICE/WEIGHT+0.5)
160     PRINT TAB(20);UNITPRICE
170     END
```

You can see how remarks can be inserted, how the column headings are displayed, and how each line of the table is formed. As soon as the heading of the table is displayed the program is ready to accept values for the cost and weight of the first box. Remember: after you have entered the weight, press return so that the computer will know that you have finished entering the data. After the price per kilogram is displayed, a question mark and the cursor will be placed at the beginning of the next line and

then you are ready to enter the data for a new box. In the program we have repeated three statements, without change, one set of three for each box. If we had 100 boxes, this would have been a little monotonous. When we want to repeat statements we do *not* do it this way; a more convenient way is possible with a new Basic statement that will cause this kind of repetition. But that comes in the next chapter.

A second kind of output labelling was already used in the previous example but can be illustrated by a program segment

```
PRICE=5
PRINT "PRICE=",PRICE
```

This would result in the output

```
PRICE=5
```

This method is easier when just a few numbers are being displayed.

PROGRAM TESTING

It is easy to make mistakes in programming. Always use a pencil when writing programs so you can erase your mistakes. Things that are crossed out are messy. The first thing you should do to test a program is to read over your program carefully to spot errors. It is valuable to trace the execution yourself before you enter it into the computer. Your goal should always be to produce programs that you *know* are correct without testing, but this is not always possible. You could ask someone else to read it too. If another person cannot understand your program it may show that your program is poorly written or has errors. Next you enter your program and proofread it to see that it is the same as what you wrote on the piece of paper.

If you have made errors in your program that involve the form of statements, the compiler spots these during running and reports them on the output. It gives a line number and notes that there is an error. Errors in form are called *syntax errors.* Examples of common syntax errors are

1. choosing a variable name that is, or contains, a keyword
2. forgetting the END
3. misspelling a keyword

In a way, a syntax error is a good error since it is detected for you by the computer. But it is frustrating to have to correct it and rerun the job. It wastes time. Some people say that having syntax errors is a symptom of sloppy programming and a sure indication that there are other errors.

If there are no syntax errors, execution can continue. This does not, however, mean that all is well.

If answers are displayed, they should be checked against hand calculated answers. If they agree, it is possible that your program is correct. If they disagree it is possible that your hand calculations are incorrect or that your program has errors. The errors now are usually of a kind called *semantic errors.* You are asking for a calculation that you did not mean to ask for. It has a different *meaning* from your intentions. For instance, you are adding two numbers and you meant to subtract them.

To find semantic errors you must look at the program again and try to trace what it must be doing rather than what you thought it would do. To help in the tracing it is sometimes necessary to insert additional PRINT instructions between other statements and output the current value of variables that are changing. In this way you can follow the machine's activity. These extra PRINT instructions can be removed after the errors have been found. This is one reason we start numbering our program lines by 10s. It leaves room for extra PRINT statements or additions to the program.

We had hoped to number all programs in the book by 10s but we made mistakes (unbelievable but true!) and you can see places where the numbering is not consistently by 10s. We could have renumbered our programs after they were corrected but decided to leave the evidence of our afterthoughts for you to see.

COMMON ERRORS IN PROGRAMS

When you try running a program on a computer, the computer may detect *errors* in your program. Since the computer does not understand the purpose of your program, its error messages are limited to calling attention to the specific illegalities which it detects. Unfortunately, the computer's error messages usually do not tell you how to correct your program so that it will solve the problem you have in mind.

In order to help you avoid such errors, we list some of the errors which commonly occur in a beginner's programs.

Missing END at the end of the program. (As it happens, in Applesoft Basic it is not necessary to have an END but we will not use this peculiarity.)

Missing quotes, especially the last quote. Consider the following erroneous statement:

```
PRINT "INVOICE
```

This statement is missing a quote after the E

Mistaking I for 1 - The characters I and 1 look similar, but are entirely different to the computer.

Mistaking O for 0 - The characters O(oh) and 0(zero) look similar, but are entirely different to the computer. The digit zero is sometimes displayed with a slash through it so that it is distinctly different from the oh.

COMMUNICATING WITH THE BASIC INTERPRETER

In the computer, variables are stored in particular locations in the memory but we need not concern ourselves with where they are stored; that is handled for us by the Basic interpreter. The interpreter itself has certain memory locations which it uses for different purposes. In the Basic language there is a way in which you can find out what the interpreter has stored in a particular location. This is done by means of the built-in function PEEK. We peek at a memory location and find its value. For example, the assignment statement

ERR=PEEK(222)

will set the variable ERR (we did not use ERROR because it contains the reserved word OR) to an integer which is the error code of the last encountered error. These codes are listed in Appendix 5. We will use this ability when we come to reading files of data from the disk. It helps us to use the error produced by trying to read beyond the last available data item to stop the reading process.

The other thing we can do is to write an integer into any machine address. This is done by a Basic instruction of the form

POKE machine address, integer

One use we will make of this is to change the size of the window on the screen that displays your text of program and data. For example, the instruction

POKE 32,9

sets the left margin of the display window 10 characters over from its normal position. The character positions are numbered from 0 to 39. The window width should be reduced before this is done. To reduce the window to 30 characters in width use

POKE 33,30

The instruction

```
POKE 33,40
```

will restore it to its normal width. The instruction

```
TEXT
```

will also do this. The instruction

```
POKE 35,12
```

sets the bottom margin of the screen so only 13 lines (0 to 12) are displayed. The normal setting is 23 giving 24 lines of text (0 to 23).

CHAPTER 5 SUMMARY

This chapter introduced *variables*, as they are used in programming languages. Essentially, a variable is a memory location, or cell, which can hold a *value*. Suppose X is the name of a variable; then X denotes a cell. The name X implies that it is a variable having the numerical type, then the cell for X can hold a number such as 9, 291, 3.56, 7.1E12, 0 or -11.

The following important terms were discussed in this chapter.

Identifier - can be used as the name of a variable. An identifier must begin with a letter; this letter can be followed by additional letters or digits. The following are examples of identifiers: X, I, WIDTH, INCOMETAX and A1. In APPLESOFT II Basic only the first 2 characters of an identifier are used, so that no two identifiers in a program should have the same first 2 characters. Identifiers of variables may not be, or contain, Basic keywords. If they do, a syntax error will be reported and you must choose another name.

Type - Each variable has a type; in this chapter we introduced the numerical type. There are, as well, variables for alphabetic information.

Constant definition - establishes named constants for use in a program. For example, the definitions

```
PI=3.14159
CMPERINCH=2.54
```

create named constants PI and CMPERINCH. Definition of constants should be immediately after the REM PROGRAM heading and precede any PRINT, INPUT, or assignment statements.

Assignment - means a value is assigned to a variable. For example, the following is an assignment statement which gives the value 52 to the variable I:

```
I=52
```

An alternative form of assignment in Basic language is

```
LET I=52
```

We will not use this form.

Truncation - throwing away the fractional part of a number. A variable Y can be given the truncated value of a positive variable X by using the INT function. The statement

```
Y=INT(X)
```

will assign the integral part of the value of the variable X to the variable Y.

Rounding - changing a real number to the nearest integer. If X contains a positive number

```
Y=INT(X+0.5)
```

assigns the value of X, rounded off, to the variable Y.

Data (or input data) - values which a program can read are typed in on the keyboard during the execution of the program.

INPUT - means "Read data." The INPUT statement reads data values into a list of variables. As data are entered the items are separated by commas.

INPUT statement - this statement is of the form:

```
INPUT list of variable names separated by commas
```

Reading will automatically proceed to the next input line when the values of one line have all been read. The last data item to be read by an INPUT statement must be followed by a return.

READ statement - similar to INPUT except that the data is presented along with the program in lines beginning with the keyword DATA instead of being entered at the time of execution from the keyboard.

Remarks - information in a program which is intended to assist a person reading the program. The following is a remark which could appear in a Basic program:

```
REM THIS PROGRAM COMPUTES GAS BILLS
```

Remarks do not affect the execution of a program. We use a remark of the form

```
REM PROGRAM name of program
```

to begin every program.

Documentation - written explanation of a program. Remarks are used in a program to document its actions.

Keyword - a word, such as INPUT or PRINT, which is part of the Basic programming language. Keywords must not be used as identifiers. These are sometimes called reserved words.

Syntax Errors - improper parts of a program. For example, the statement

```
PRINT "HELLO
```

has a syntax error in that the second quote is missing. If the computer detects an error in your program, it will output an error message.

PEEK - a built-in function that allows you to obtain the value stored in any machine address. The value of

```
PEEK(222)
```

in particular is the error code stored in location 222 by the Basic interpreter.

POKE - a instruction which allows you to store an integer in a particular machine address. If this is an address used by the Basic interpreter we can cause certain results to occur. The instruction

```
POKE 33,30
```

sets machine address 33 to the integer 30. Location 33 is the one the interpreter uses to store the length of the line of characters displayed on the screen. Normally location 33 has the value 40

CHAPTER 5 EXERCISES

1. Suppose that I, J and K are variables that presently have the values 5, 7 and 10. What will be displayed as a result of the following statements?

```
PRINT I,I+1,I+J,I+J*K
K=I+J
PRINT K
J=J+1
PRINT J
I=3*I+J
PRINT I
```

2. RADIUS, DIAMETER, CIRCUMFERENCE, and AREA are variables. A value has been read into RADIUS via an INPUT statement. Write statements which do each of the following.

   (a) Give to DIAMETER the product of 2 and RADIUS.

   (b) Give to CIRCUMFERENCE the product of pi (3.14159) and DIAMETER.

   (c) Give to AREA the product of pi and RADIUS squared. (RADIUS squared can be written as RADIUS*RADIUS or RADIUS^2.)

   (d) Output the values of RADIUS, DIAMETER, CIRCUMFERENCE and AREA.

3. Suppose I is a variable that has already been given a value via an assignment statement. Write statements to do the following.

   (a) Without changing I, output out twice the value of I.

   (b) Increase I by 1.

   (c) Double the value of I.

   (d) Decrease I by 5.

4. What will the following statements cause the computer to output?

```
M=43
N=211
P=M
M=N
N=P
PRINT M,N
```

5. (a) What will be displayed by the following program if 22 and 247 are input? How about -16 and 538?

```
010     REM PROGRAM PAIRS
020     PRINT "INPUT TWO INTEGERS"
030     INPUT FIRST,SECND
040     PRINT FIRST+SECND
050     INPUT FIRST,SECND
060     PRINT FIRST+SECND
070     END
```

6. (a) What will be displayed by the following program if the grades are 81.7 and 85.9?

```
010    REM PROGRAM COMBINE
020    REM CALCULATE TERM MARK
030    PRINT "INPUT TWO GRADES"
040    INPUT G1,G2
050    MARK=INT((G1+G2)/2+0.5)
060    PRINT MARK
070    END
```

7. Trace the execution of the following program. That is, give the values of the variables LNGTH, WIDTH, and ABOUT and give any output after each line of the program. Use as the two input values 9.60 and 15.9.

```
010    REM PROGRAM AREA
020    REM INPUT SIZES CONVERT FEET TO YARDS
030    INPUT "WIDTH="; SIZE
040    WIDTH=SIZE/3
050    INPUT "LENGTH="; SIZE
060    LNGTH=SIZE/3
070    ABOUT=INT(WIDTH*LNGTH+0.5)
080    PRINT "LENGTH AND WIDTH ARE:";
           LNGTH,WIDTH
090    PRINT "AREA IS:";LNGTH*WIDTH,
         " THIS IS ABOUT ";ABOUT;" (SQ YARDS)"
100    END
```

8. Write a program which reads three values and outputs their average, rounded to the nearest whole number. For example, if 20, 16, and 25 are entered then your program should output 20. Make up your own data for your program.

9. Write a program which reads a weight given in pounds and then outputs the weight in (1) pounds, (2) ounces, (3) kilograms and (4) grams. Note: 16 ounces equal one pound, 2.2046 pounds equal one kilogram and 1000 grams equal one kilogram. Use named constants in your program.

Chapter 6

CONTROL FLOW

So far in Basic we have learned to write programs with statements that cause the computer to read data and assign values to variables, evaluate arithmetic expressions and assign the values to variables, and output results with labels. In all programs the statements were executed in sequence until the END was reached, at which time the program was terminated. In this chapter we will learn two ways in which the order of executing statements may be altered. One involves the repetitious use of statements; the other involves a selection between alternate paths in the flow of statements. The first is called a *loop* or *repetition*, the second a *selection*. We speak of the *flow of control* since it is the control unit of the computer that determines which statement is to be executed next by the computer.

COUNTED LOOPS

The normal flow of control in a program is in a straight line. So far in all our programs, one statement is executed after another. We can, however, give a statement that will cause statements that follows it to be repeated. In the last chapter, in the example where we were computing information about boxes of soap flakes, we had to write the statements over and over to get repetition. A statement that will produce repetition is the *counted FOR loop.* For our example we could have written

```
FOR I=1 TO 3
    INPUT PRICE,WEIGHT
    UNITPRICE=INT(PRICE/WEIGHT+0.5)
    PRINT UNITPRICE
    NEXT
```

The three statements that we had to repeat three times are prefaced by

```
FOR I=1 TO 3
```

and followed by NEXT

The three statements bracketed by the FOR and NEXT we call the *body* of the loop. The variable I is an index which counts the number of repetitions. First the index I is set to 1, then the body of the loop is executed. After the execution, control is sent back to the FOR. At this time the index I is increased by 1, making it 2. The body is again executed and, then, back we go to the FOR. This time I becomes 3 and a third execution of the body in the *FOR loop* takes place. When control returns to the FOR this time, I is found be already equal to the final value 3 so control goes out of the loop to the statement after the NEXT statement.

A counted or *indexed* FOR loop is used whenever we know exactly how many repetitions we want to take place. We do not need to start the count at 1. We could, for instance, write:

```
FOR COUNT=12 TO 24
```

Here we have called the index COUNT and are starting at 12 and going up to, and to include, 24.

In these statements we have been counting forward by 1. It is possible to count by any step size we want. This statement causes counting by 2s:

```
FOR COUNT=2 TO 20 STEP 2
```

Here the index COUNT is set to 2, 4, 6... up to 20; there are 10 repetitions. We can also count backwards, say by -1. If we write

```
FOR COUNT=10 TO 1 STEP -1
    statements
    NEXT
```

it will cause the "statements" to be executed with COUNT taking the values 10, 9, 8, ..., 1.

If we have a FOR statement of this type

```
FOR I=1 TO N
```

the loop will execute a number of times which depends on the value of the variable N. If N is less than 1, say 0, the loop will still be executed once. If N is not an integer the result will be the same as if we used INT(N) for the upper limit. The value of N must not be altered in the loop body.

The other kind of loop will be called the REPEAT UNTIL loop but we cannot introduce it until we look at *conditions.* The REPEAT UNTIL loop causes repetition until a certain condition is true. The condition concerned is written at the beginning of the loop.

CONDITIONS

There are expressions in Basic that are called conditions or *relational expressions* and these have values that are either *true* or *false*. The following is a list of relational expressions with their value written on the same line. The symbol > means is greater than, < means is less than, the equal sign means is equal to, and the sign <> means is not equal to. The sign >= means is greater than or equal to.

relational expression	value
5=2+3	true
7>5	true
2<6	true
5+3<2+1	false
6<>10	true
5>5	false
5>=5	true

You can see how these work. These are sometimes called Boolean expressions after the logician George Boole.

There are also compound conditions formed by taking two single conditions and putting either the *Boolean (logical) operator* AND or the Boolean operator OR between them. When conditions are compounded in this way, it is sometimes clearer if each simple condition has parentheses around it.

With AND *both* conditions must be true or else the compound condition is false. For example,

(8>7) AND (6<3)

is false since (6<3) is false. With OR, if *either* or *both* of the single conditions is true, the compound condition is true. For example, (8>7) OR (6<3) is true since (8>7) is true. It is possible to have multiple compoundings. For example,

((8>7) AND (2=1+1)) AND ((6>7) OR (5>1))

is true. The parentheses here show the sequence of the operations.

A Boolean operator that requires only one condition is the NOT operator. The condition

NOT(5>6)

is true since 5 is *not* greater than 6.

CONDITIONAL LOOPS

We have introduced the notion of a condition; now we will actually use it. One of the major uses of conditions is in the conditional loop: the REPEAT UNTIL condition loop. Unfortunately there is no such statement in Basic. We must construct the statement out of two other Basic instructions that are available. One of these is the statement

GOTO line number

which causes the instruction in the line whose number is given in the GOTO statement to be executed next. This Basic statement causes the normal sequential execution of Basic instructions to be broken. We say control is sent to the instruction with the designated line number and continues from there.

The other Basic instruction that is used to construct a conditional loop is the statement

IF condition THEN statement

When this instruction is executed the condition is evaluated and if it is found to be true then the statement after the keyword THEN is executed. If the condition is false, the statements that follow in the next line after the IF statement are executed.

Perhaps the easiest thing to do now is to illustrate the REPEAT UNTIL conditional loop by an example.

In the following program a REPEAT UNTIL loop is used to accomplish what the counted FOR loop did for the soap flakes boxes.

```
010    REM PROGRAM SOAP2
020    REM COMPUTE UNIT PRICE
030    REM PRINT HEADINGS OF TABLE
040    PRINT"     PRICE ";"WEIGHT(KG)";" PRICE/KG"
050    I=1
060    REM REPEAT UNTIL I>3
070    IF I>3 THEN GOTO 130
080        INPUT PRICE,WEIGHT
090        UNITPRICE=INT(PRICE/WEIGHT+0.5)
100        PRINT TAB(20); UNITPRICE
110        I=I+1
120        GOTO 070
130    REM END OF LOOP
140    END
```

In statement 050 the value of the variable I appearing in the condition is set initially to 1, then we enter the loop. This stage is called *initialization.*

In line 060 we begin the loop with a remark. The condition after the IF is false since I is 1, which is less than 3. Thus the statements going down to line 120 are executed. These statements constitute the *body* of the conditional loop. In the body, statement 110 *alters* the value of the variable appearing in the condition. This means that the variable's value is changing each time around the loop. At the end of the first execution of the loop it becomes 1+1=2. After the loop body has been executed, the GOTO in line 120 sends control to the start of the loop. The condition is then examined a second time and since it is false (2<=3), the body is executed a second time. It will be false also on the third time but, on the fourth round, I will be 4 and (4>3) is true. This means that on the fourth round the statement after the THEN will be executed. This is GOTO 130. Control is sent to line 130 which is a remark indicating that the looping is finished.

The various phases of a REPEAT UNTIL conditional loop are

Phase 1. Initialization, especially of the variable in the condition

Phase 2. Test condition and if false then go to the next statement which is the beginning of the body of the loop, if true go to the statement following the GOTO that ends the body

Phase 3. Execute the statements that constitute the body of the loop which include altering the variable in the condition

Phase 4. Return to phase 2.

Phase 1 is necessary to give the variable appearing in the condition an initial value. The body of the loop must alter the variable in the condition in addition to taking some other action.

We have identified the conditional loop by having the first statement of the form

REM REPEAT UNTIL condition

and the last statement

REM END OF LOOP

This is to make our program more understandable. The body of the loop, including the terminating GOTO, is indented for the same reason. These are conventions that we have adopted to help you to write better programs.

UNCONDITIONAL LOOP

An unconditional loop is a loop that has no stopping condition; it loops forever. It is sometimes called an *infinite loop.* We showed an example of such a loop when we first introduced programming. Here is another

```
010     REM PROGRAM FOREVER
020     REM INFINITE LOOP
030         PRINT "THIS IS MONOTONOUS"
040         GOTO 030
050     END
```

The loop body begins in line 030; there is no statement about the stopping condition. The end of the loop body is in line 040 where the GOTO sends control back to the beginning. We do not have a remark after line 040 saying

```
REM END OF LOOP
```

because there is no programmed end of the loop; it loops forever. The END in line 050 merely says that this is the last statement in our program.

Of course we do not really let our loop run forever; we stop it by typing CTRL-C. If we then want it to start up again, we type CONT for continue; but we can just go on with something else. The only time we ever program an infinite loop is when we are ourselves interacting with the computer and we want to choose the time to stop it. We provide the stopping condition externally to the program. It is sometimes easier to do this than program a stopping condition.

READING INPUT

As an example of looping we will look at reading data for student marks and computing an average mark for the class. The number of students in the class is not known at the time the program is prepared. The only real problem will be to stop reading marks after the last mark is entered and compute the average. There are two distinct ways of doing this. One is to count the number of students and enter the count before you begin entering the marks. Then we use a counted FOR loop to read them. The second method is to enter at the end of the data a piece of data that is impossible as a real entry. We call it a *dummy entry.* Sometimes it is called an *end-of-file marker.*

We will now examine the two methods in turn. Suppose, to talk specifically, that each data entry consists of a student number and a grade received in an examination. To illustrate we will have only three data entries, but you can see how it will work with more.

Method 1. Counting the number of entries

```
010     REM PROGRAM MARKS1
020     INPUT "COUNT=";  COUNT
030     SUM=0
040     PRINT " STUDENT";TAB(12);"MARK"
050     FOR I=1 TO COUNT
060         INPUT STUDENTNUMBER,MARK
070         SUM=SUM+MARK
080         NEXT
090     PRINT " AVERAGE=";INT(SUM/COUNT+0.5)
100     PRINT " COUNT=";COUNT
110     END
```

The screen will display

```
COUNT=3
  STUDENT      MARK
?1026,           86
?2051,           90
?3163,           71
  AVERAGE=82
  COUNT=3
```

Notice that we took the trouble to label the output.

In the second method we will place a dummy entry with two zeros in it after we have finished all the valid entries. This will then trigger the calculation of the average mark.

Method 2. Testing for the dummy entry

```
010     REM PROGRAM MARKS2
020     COUNT=0
030     SUM=0
040     PRINT " STUDENT";TAB(12);"MARK"
050     INPUT STUDENTNUMBER,MARK
060     REM REPEAT UNTIL STUDENTNUMBER=0
070     IF(STUDENTNUMBER=0)THEN GOTO 120
080         SUM=SUM+MARK
090         COUNT=COUNT+1
100         INPUT STUDENTNUMBER,MARK
110         GOTO 070
120     REM END OF LOOP
130     PRINT " AVERAGE=";INT(SUM/COUNT+0.5)
140     PRINT " COUNT=";COUNT
150     END
```

Here the display on the screen is

```
  STUDENT      MARK
?1026            86
?2951            90
?3163            71
?  0              0
  AVERAGE=82
  COUNT=3
```

In this example you will notice that the initialization involves reading the first input line outside the loop, in order to get a value for the variable STUDENTNUMBER appearing in the condition of the REPEAT UNTIL. Since the first line has already been read, its mark must be added to the sum before a new line is read. This means that the sequence is SUM=SUM+MARK then INPUT, rather than the way it is in method 1. As soon as the new line has been read, we return to the IF where the condition is tested.

Methods 1 and 2 for dealing with a variable number of items are used again and again in programming. The REPEAT UNTIL is more difficult to program but probably more useful, since if there are many items, it is better for the user to stick in an end-of-file item than to count items.

EXAMPLES OF LOOPS

We will now give example programs to illustrate details about loops. The examples each draw a zigzag. Here is the first example:

```
010    REM PROGRAM WIGGLE
020    FOR J=1 TO 3
030        PRINT "       *"
040        PRINT "        *"
050        PRINT "         *"
060        PRINT "        *"
070        NEXT
080    END
```

This program, appropriately called WIGGLE, outputs the following pattern:

```
       *
        *
         *
        *
       *
        *
```

```
  *
 *
*
 *
  *
 *
```

The WIGGLE program causes the body of the loop, lines 030 through 060, to be executed three times. The variable J is 1 during the time the first four stars are displayed. J is 2 during the time the next four stars are displayed, and J is 3 while the last four stars are displayed. After the last star is displayed, J is set to 4 and since J then exceeds the limiting value, 3, of the loop, the loop is terminated.

Notice that in this program the variable J is used for only one purpose: to see that the loop is repeated the desired number of times. Line 020 means, essentially, "Repeat this loop three times." If we replaced line 020 by the following line

```
FOR J=9 TO 11
```

then the program would still output the same pattern. The only difference is that J would have the values 9,10, and 11 during the output of the stars and would end up with the value of 12. Although this replacement for line 020 does not change the pattern displayed, it should not be used because it makes the program more confusing for people to understand. This is because people more naturally think of "repeat this loop three times" as running through the loop with values 1, 2, and 3, rather than values 9, 10, and 11.

Here is one more possible replacement for line 020 which does not change the displayed pattern:

```
FOR J=3 TO 1 STEP -1
```

In this case, J will be 3 while the first four stars are displayed, then J will be 2 while the next four stars are displayed, and then J will be 1 while the last four stars are displayed. Finally, J will end up with the value of zero. This illustrates the fact that if the step size, which is -1 here, is negative, then the loop will count backwards to smaller values. Again, for this example program, the original version of line 020 is preferable because it is easier to understand its meaning at a glance.

Once a FOR loop finishes, there is some confusion about the final value of the counting variable. You should avoid this confusion by following this advice:

> When a FOR loop has finished, do not use the final value of the counting variable.

There is another possible source of confusion in FOR loops. For example, what happens in WIGGLE if we set the counting variable J to 15 by an assignment statement in the loop body? The result is that J is not in the range 1 to 3. The FOR loop no longer means: repeat for J equal to 1 then 2 then 3. To avoid this confusion, Basic has this rule:

> Inside a counted FOR loop, the program must not alter the value of the counting variable.

We will now rewrite our WIGGLE program using REPEAT UNTIL instead of a counted FOR. We will call our new program WAGGLE. (Did you know that in German "wiggle waggle" means "waddle" like a duck? Well it does.)

```
010    REM PROGRAM WAGGLE
020    J=1
030    REM REPEAT UNTIL J>3
040    IF J>3 THEN GOTO 110
050        PRINT "       *"
060        PRINT "        *"
070        PRINT "         *"
080        PRINT "        *"
090        J=J+1
100        GOTO 040
110    REM END OF LOOP
120    END
```

This WAGGLE program works like our previous WIGGLE program. Lines 020, 030, 040, and 090 of WAGGLE are equivalent to line 020 of WIGGLE. Since it is easier to see that line 020 of WIGGLE means, "Repeat this loop three times," the WIGGLE version is preferable. We will, however, use WAGGLE to illustrate a few more points about loops.

In the WAGGLE program, consider moving line 090, which is

```
J=J+1
```

up to between lines 040 and 050. This change does not alter the displayed pattern. It simply changes the point at which J has its value increased. J will have the value 2 while the first four stars are displayed, then 3 while the next four stars are displayed and finally 4 while the last four stars are displayed. J ends up with the value of 4. Even though J is set to 4 before the last four stars are displayed, the loop is not stopped. This is because J is not compared to the limit value 3 until control returns to line 040. This illustrates the fact that in a REPEAT UNTIL loop, the condition is tested only once - at the top - each time through the loop.

Now let us look back at the WAGGLE program. Suppose that you prepared this program for the computer and mistakenly made line 090 into

```
J=J-1
```

The mistake is that the plus sign was changed to a minus sign. Such a small mistake! Surely the computer will understand that a plus was wanted! But it will not do so. The computer has a habit of doing what we *tell* it to do rather than what we *want* it to do. Given the WAGGLE program, with the mistake, the computer will do the following. With J set to 1 it will output the first four stars. Then, as a result of the erroneous line 090, it will set J to 0 and will output another four stars. Then it will set J to -1 and output four more stars. Then it will set J to -2 and output four more stars and so on and so on. In theory, it will *never stop* outputting stars because the condition J>3 will always be false. This is an accidental *infinite loop.* You will have to stop the computer from executing your program by pressing the CTRL-C.

BRANCHES IN CONTROL FLOW

We have learned how to change from a flow of control in a straight line, or *sequential* control, to flow in a loop, either counted or conditional. Now we must look at a different kind of structure in the sequence of control. This structure is called selection or *branching.* It is a little like a fork in the road where there are two paths that can be followed. The road branches into two roads. When you come to a fork in a road you must *decide* which of the two branches you will take. Your decision is based on where you are heading. Suppose one sign at the fork gives the name of your destination and the other road sign gives some other name. Suppose your destination is Toronto; an instruction for deciding which branch to take might be

```
IF LEFTBRANCHSIGN="TORONTO" THEN
    take the left branch
ELSE
    take the right branch
```

We have written this decision in the form that is used in a number of high-level languages such as Pascal, PL/1, or Fortran 77. This is the ideal form from the point of view of structured programming. Unfortunately Applesoft Basic does *not* have this particular statement in the language so that, just as with conditional loops, we are required to construct it out of what is available. Again we will use the Basic statements IF...THEN and GOTO to make an IF...THEN...ELSE construct.

Suppose that there is a variable called ACLASS which contains the number of students in a class called A. Students are to be assigned to Class A if their mark in computer science (CSMARK) is over 80; otherwise they are to be assigned to Class B (BCLASS). The IF...THEN...ELSE type statement which decides which class to place the student in, and counts the number going into each class, is

```
IF CSMARK>80 THEN
    ACLASS=ACLASS+1
ELSE
    BCLASS=BCLASS+1
```

The IF...THEN...ELSE causes control to split into two paths but, unlike forks in roads, it immediately comes back together again. This means that we are never in any doubt about what happens; after the execution of one or the other of the two branches, the THEN branch or the ELSE branch, the control returns to the normal sequence. One way of looking at the IF...THEN...ELSE is that it provides two possibilities, only one of which is to be *selected*, depending on whether the condition following the IF is true or false. After one or the other path is executed the normal control sequence is resumed. Remember: this IF...THEN...ELSE that we have just shown is *not* a part of Applesoft Basic.

Now we come to the problem of constructing an IF...THEN...ELSE in Applesoft Basic. Here is the set of statements which we will use to substitute for the example given:

```
100     REM IF CSMARK>80 THEN...ELSE
110     IF CSMARK>80 THEN GOTO 150
120         REM DO ELSE
130         BCLASS=BCLASS+1
140         GOTO 170
150         REM DO THEN
160         ACLASS=ACLASS+1
170     REM END OF SELECTION
```

Since we have already met the two Basic statements IF...THEN and GOTO you can follow the program segment and see that it is equivalent to the IF...THEN...ELSE of many other high-level programming languages. Notice that we have remarks at the beginning and end of the whole structure and at the start of the THEN body and the ELSE body. Also we have indented the structure so that you see the extent of the selection construct easily. In the remark that begins the IF...THEN...ELSE construct we have written the condition. This condition is repeated in the next line which is a Basic IF...THEN statement. You do not have to write out the condition twice especially if it is a compound condition. The remark could be simply

REM IF...THEN...ELSE

The main reason for having the remark is to make sure when we see an IF...THEN Basic statement in a program we know whether it is being used as part of a loop or a selection construct.

When the selection construct has no ELSE clause we can make it directly out of the Basic IF...THEN and do not need to include the beginning and ending remarks. The form is

IF condition THEN statements of "THEN" body separated by colons

If the THEN body consists of one statement it is very simple; if there are more statements than one in the THEN body they must be separated by colons and all placed on the same line with the IF. Do not press return until you have typed the last statement of the THEN body. The display on the screen may move to a new line but this *physical line* does not have anything to do with the *logical line* which you terminate by typing return. Lines in Basic programs are limited in length to 239 characters which includes all necessary spaces so that large THEN bodies cannot be handled in this way. If there are too many statements in the THEN body to fit into a single logical line then you must revert to the form we have shown for the IF...THEN...ELSE construct. The body of the ELSE will just be the GOTO in that case.

THREE-WAY BRANCHES

We have seen how a sequential control structure can be split up into two branches and then brought together again. What do we write if we have a situation where more than two branches are required? We will do a three-way branch and then you will see how any number of branches can be achieved. This can also be viewed as selecting one of three alternatives.

As an example, we will write a program that counts votes in an election. Suppose that there are three political parties called Conservative (right), Radicals (left), and Mugwumps (middle-roaders) and that to vote for one of these parties you enter a 1, or a 2, or a 3 respectively. Here is the program that reads the vote entries and counts each party and the total. The last vote entry is to be the dummy value -1.

```
010    REM PROGRAM VOTING
020    CNSERVTIVE=1
030    RADICAL=2
040    MUGWUMP=3
050    RIGHT=0
060    LEFT=0
070    MIDDLE=0
080    PRINT "ENTER VOTES"
090    INPUT VOTE
100    REM REPEAT UNTIL VOTE=-1
110    IF VOTE=-1 THEN GOTO 290
120        REM IF VOTE=CNSERVTIVE THEN...ELSE
130        IF VOTE=CNSERVTIVE THEN GOTO 240
140            REM DO ELSE
150            REM IF VOTE=RADICAL THEN...ELSE
160            IF VOTE=RADICAL THEN GOTO 200
170                REM DO ELSE
180                IF VOTE=MUGWUMP THEN MIDDLE=MIDDLE+1
190                GOTO 220
200                REM DO THEN
210                LEFT=LEFT+1
220            REM END OF SELECTION
230            GOTO 260
240            REM DO THEN
250            RIGHT=RIGHT+1
260        REM END OF SELECTION
270        INPUT VOTE
280        GOTO 110
290    REM END OF LOOP
300    COUNT=RIGHT+LEFT+MIDDLE
310    PRINT "CONSERV.";TAB(10);"RADICAL";TAB(20);
                "MUGWUMP";TAB(30);"TOTAL"
320    PRINT RIGHT;TAB(10);LEFT;TAB(20);MIDDLE;TAB(30);COUNT
330    END
```

In this program we have two different things happening. One is a three-way branch, which is accomplished by a series of three IF...THEN...ELSE constructs, one for each value of VOTE. This means that an invalid vote does not get counted anywhere. The third construct degenerates into an IF...THEN because there is no ELSE. (We did not use the variable identifier CONSERVATIVE because it contains two Basic keywords ON and AT.)

The minimum program needed for a three-way branch would be one IF...THEN...ELSE construct nested inside another. We will show this as if we have an IF...THEN...ELSE without having to construct one, as we do in Basic, out of IF...THEN and GOTOs.

```
IF VOTE=CNSERVTIVE THEN
    RIGHT=RIGHT+1
ELSE
    IF VOTE=RADICAL THEN
        LEFT=LEFT+1
    ELSE
        MIDDLE=MIDDLE+1
```

If there are any invalid votes, they are given to the MUGWUMP party.

There is no fixed way of doing the job. Here is another try at it.

```
IF VOTE<MUGWUMP THEN
    IF VOTE<RADICAL THEN
        RIGHT=RIGHT+1
    ELSE
        LEFT=LEFT+1
ELSE
    MIDDLE=MIDDLE+1
```

Again we have nesting of two IF...THEN...ELSE constructs, but in a different sequence. It is good practice not to have a second IF...THEN...ELSE nested after the THEN it is preferable to have it in the ELSE branch.

EXAMPLES OF SELECTION

We will now give a series of examples of IF...THEN...ELSE constructs which might be used in a government program for handling income tax. Let us suppose that the program is to write notices to people telling them whether they owe tax or they are to receive a tax refund. The amount of tax is calculated and then the following is executed. Again we use the IF...THEN...ELSE notation which must be translated into the equivalent Basic construct.

```
IF TAX > 0 THEN
    PRINT " TAX DUE IS ";TAX;" DOLLARS"
ELSE
    PRINT " REFUND IS ";-TAX;" DOLLARS"
```

Notice that it was necessary to change the sign of "tax" when we output the refund.

Unfortunately, our program, like too many programs, is not quite right. If the calculated tax is exactly zero, then the program will output REFUND IS 0 We could fix this problem by the following.

```
IF TAX > 0 THEN
    PRINT " TAX DUE IS ";TAX;" DOLLARS"
ELSE
    IF TAX = 0 THEN
        PRINT " YOU OWE NOTHING"
    ELSE
        PRINT " REFUND IS ";-TAX;" DOLLARS"
```

We have used a *nested* IF...THEN...ELSE construct to solve the problem, that is, an IF...THEN...ELSE construct which is inside an IF...THEN...ELSE construct. In general, we can nest any kind of statement or construct inside an IF...THEN...ELSE construct including assignment statements, INPUT and PRINT statements, and FOR statements.

Now suppose that when the tax is due we wish to tell the taxpayer where to send his check. We can expand the program as follows:

```
IF TAX > 0 THEN
    PRINT " TAX DUE IS ";TAX;" DOLLARS"
    PRINT " SEND CHECK TO DISTRICT OFFICE"
ELSE
    IF TAX =0 THEN
        PRINT " YOU OWE NOTHING"
    ELSE
        PRINT " REFUND IS ";-TAX;" DOLLARS"
```

MULTI-WAY BRANCHES

We have shown how IF THEN...ELSE constructs may be nested to produce three-way branching or four-way branching. There is another method of branching when there are more branches than two that is often in a high-level language. It is called the CASE construct. This is not available in Basic but another kind of statement is available. It has the form

ON case number GOTO line for case 1, line for case 2,... line for case *n*

Following this statement there would be *n* different branches or cases which are numbered from 1 to *n*. The case number specifies which of the *n* possibilities will be executed.

Suppose, for example, we had a program to read 20 numbers, which must be between 1 and 4 inclusive, that are to be input and which prints out how many of each one of the numbers there are. Here is the program

```
010    REM PROGRAM FOURWAY
020    REM DEMONSTRATE A MULTI-BRANCH SITUATION
030    WUN=0
040    TWO=0
050    THREE=0
060    FOUR=0
070    PRINT "ENTER 20 NUMBERS BETWEEN 1 AND 4 INCL."
080    FOR I=1 TO 20
090        INPUT NUMBER
100        REM CASE CONSTRUCT
110        ON NUMBER GOTO 120, 140, 160, 180
120            WUN=WUN+1
130            GOTO 190
140            TWO=TWO+1
150            GOTO 190
160            THREE=THREE+1
170            GOTO 190
180            FOUR=FOUR+1
190        REM END OF CASE
200        NEXT
210    PRINT WUN;" ONES ";TWO;" TWOS ";THREE;" THREES ";FOUR;" FO
220    END
```

Notice that we prefaced the ON statement with a remark to indicate that it is a case construct. If NUMBER is a one we go to line 120 where we increase the variable WUN by 1; then line 130 says GOTO 190 which is the end of the case construct. We have indented the body of the case construct so that it is easy to spot.

PARAGRAPHING THE PROGRAM

In order to follow the *structure* of the nesting of IF...THEN...ELSE constructs we have indented the program so that the ELSE and THEN bodies that belong to each other are lined up vertically and are indented from the beginning and ending remarks and the IF statement. This is called *paragraphing* the program, and is analogous to the way we indent paragraphs of prose to indicate grouping of thoughts. Paragraphing makes a valuable contribution to understandability and is a *must* in structured programming.

We will use paragraphing in the Basic programs shown in the book to help you see the structure. Unfortunately when you enter a Basic program into your computer it is automatically turned into a form with no

paragraphs even though you go to the trouble of putting indentations between the line number and the beginning of the statement. Just try entering a program with paragraphing, then type LIST and you will see for yourself. Your paragraphs have all disappeared. This may be necessary to make the interpretation of Basic statements easier for the computer but it destroys one of the most valuable devices used in structured programming.

Even so, if you get used to writing programs with paragraphing, you will be able to see the structure in your mind better even when the computer eliminates it. Maybe some day Applesoft Basic will be improved to let you keep your paragraphs in programs stored in the computer.

If you examine the programs as they appear in the book with FOR loops you will see that the loop body has been indented starting right after the FOR. We indent the statements up to and including the NEXT statement to show the scope of the loop body. In the next chapter we will be examining the situation where FOR loops are nested, and then we will use two levels of indentation.

TRACING CONTROL FLOW

We have discussed tracing the execution of a program where the values of variables were calculated by hand after the execution of each instruction. The example we showed was for a linear sequence of statement execution. When control structures include loops and branches, tracing the actual sequence of execution of the instructions is also important. We can use the computer to help us do this. If we give the command

```
TRACE
```

before we RUN the program the computer will output the sequence of the line numbers of statements executed. It will also output any data that is in PRINT statements. These two will be in the sequence that they occur. Here is an example program being traced

```
010     REM PROGRAM SEQUENCE
020     REM DEMONSTRATES TRACE COMMAND
030     PRINT "BEGIN"
040     FOR I=1 TO 3
050         PRINT I
060         NEXT
070     PRINT "END"
080     END
TRACE
RUN
```

Here is the output:

```
#10    #20    #30    BEGIN
#40    #50    1
#60    #50    2
#60    #50    3
#60    #70    END
#80
```

Lines 10, 20, and 30 are executed and, as a result of the execution of 30, BEGIN is printed. The return follows the BEGIN. If there were no instruction 30 the first line of the trace would be

```
#10 #20 #40 #50 1
```

To stop tracing programs we must give the command

```
NOTRACE
```

CHAPTER 6 SUMMARY

In this chapter we introduced statements which allow for (a) repetition of statements and (b) selection between different possibilities. We introduced *conditions* which are used to terminate the repetition of statements and to choose between different possibilities. Comparisons and Boolean or logical operators are used in specifying conditions. The following important terms were discussed in this chapter.

Loop - a programming language construct which causes repeated execution of statements. In our version of Basic loops are either unconditional (infinite) loops, counted FOR loops, or REPEAT UNTIL conditional loops.

Counted FOR loop - has the following form:

```
FOR variable = start TO limit
    statements
    NEXT
```

The variable, called the counting variable or index, is treated as an integer. Each of "start," and "limit" can be expressions; these expressions are evaluated before the repetition starts and are not affected by the statement of the loop body. Counting proceeds by 1s from the start up to and including the limit. If start is not less than or equal to limit one execution of the loop body will take

place. Counting backwards by -1 is accomplished by using the form

```
FOR variable=start TO limit STEP -1
    statements
    NEXT
```

Any step size may be specified but +1 or -1 are the most frequently used.

In Basic it is possible to follow the reserved word NEXT by the name of the variable controlling the counting. This actually slows down the execution of programs so we will *never* use it.

REPEAT UNTIL loop - has the following form in our version of Basic:

```
          REM REPEAT UNTIL condition
line A    IF condition THEN GOTO line B
              statements of loop body
              GOTO line A
line B    REM END OF LOOP
```

We indent the body of the loop and the GOTO that terminates it in the programs in this book.

Loop body - the statements that appear inside a loop.

Comparisons - used in conditions. For example, comparisons can be used in a condition to determine how many times to execute a loop body. The following are used to specify comparisons.

<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
=	equal
<>	not equal

Conditions - are either true or false. Conditions can be made up of comparisons and the following three Boolean or logical operators.

AND
OR
NOT

Conditional loop - a loop such as the REPEAT UNTIL construct which uses a stopping condition.

Unconditional loop - a loop which has no stopping condition. It would loop forever if it were not stopped by some action at the keyboard such as pressing CTRL-C. The form of such a loop is

```
          REM INFINITE LOOP
line A    body of loop
          GOTO line A
```

End-of-file (or end-of-data) detection - When a loop is reading a series of data items, it must determine when the last data item has been read. This can be accomplished by first reading in the number of items to be read and then counting the items in the series as they are read. It can also be accomplished by following the last data item by a special dummy entry which contains special, or dummy data. The program knows to stop when it reads the dummy data.

IF...THEN...ELSE construct - has the following form in many high-level languages such as Pascal or PL/1:

```
IF condition THEN
    statements forming body of THEN
[ELSE
    statements forming body of ELSE]
```

The square brackets are shown around the ELSE clause to show that it can be omitted. If the condition is true, the body of the THEN is executed. If the condition is false, the body of the ELSE, if present, is executed. In either case, control then goes to next statement after the IF...THEN...ELSE construct. The body of the THEN or ELSE can contain any statement type including an IF...THEN...ELSE construct. We try to use a second IF...THEN...ELSE only in the ELSE body.

The form we use in our better Basic to create the IF...THEN...ELSE construct is:

```
                REM IF condition THEN...ELSE
                IF condition THEN GOTO line A
                    REM DO ELSE
                    body of ELSE
                    GOTO line B
line A              REM DO THEN
                    body of THEN
line B          REM END OF SELECTION
```

If there is no ELSE and the body of the THEN is a single statement we use the much simplified form

IF condition THEN single statement of THEN body

This form does not include any beginning and ending remarks as it is perfectly clear. It is possible to program an ELSEless construct this way even when the THEN body contains several statements. The statements of the THEN body must be placed after the THEN on the same line and separated by colons. For example,

```
IF I<3 THEN I=I+1: PRINT "I=";I
```

As you type such a line the display may finish one line on the screen and automatically start a new line. This does not mean that the statement is not in one logical line in your Basic program. You must press the return to produce a new logical line. A logical line in Basic is limited in length to 239 characters so that it is not possible to write the THEN body in this way if it contains many statements.

Case construct - implemented in our version of Basic using the statement

ON case number GOTO list of line numbers separated by commas

The case number must have a value between 1 and *n* inclusive when there are *n* line numbers in the list after the keyword GOTO. The form of the CASE construct for three alternatives is

```
                REM CASE CONSTRUCT
                ON case number GOTO line A, line B, line C
line A              body of case 1
                    GOTO line D
line B              body of case 2
                    GOTO line D
line C              body of case 3
line D          REM END OF CASE
```

Paragraphing - indenting a program so that its structure is easily seen by people. The statements inside loops and inside IF...THEN...ELSE

constructs are indented to make the overall program organization obvious. The computer ignores paragraphing and, in fact, automatically eliminates it when storing and executing programs. We will use it in this book because it is such a powerful tool of structured programming and, at least, helps to make the programs in the book clearer. Perhaps it will be included in Applesoft Basic in the future.

Tracing the control sequence - by giving the system command TRACE before running a program, the sequence of line numbers of statements executed is displayed on the screen. Any output produced by PRINT statements in the program is interspersed with the line numbers of the trace. To stop tracing programs use the command NOTRACE.

CHAPTER 6 EXERCISES

1. Suppose I and J are variables with values 6 and 12. Which of the following conditions are true?

```
(a)   2*I<=J
(b)   2*I-1<J
(c)   (I<=6) AND (J<=6)
(d)   (I<=6) OR (J<=6)
(e)   (I>0) AND (I<=10)
(f)   (I<=12) OR (J<=12)
(g)   (I>25) OR ((I<50) AND (J<50))
(h)   (I<>4) AND (I<>5)
(i)   (I<4) OR (I>5)
(j)   NOT(I>6)
```

2. The following program predicts the population of a family of wallalumps over a 2-year period, based on the assumption of an initial population of 2 and a doubling of population each 2 months. What does the program output?

```
010   REM PROGRAM EXPLODE
020   FAMILY=2
030   PRINT "    MONTH"," POPULATION"
040   FOR MNTH=0 TO 24
050       PRINT MNTH;TAB(10);FAMILY
060       FAMILY=2*FAMILY
```

```
070        NEXT
080    END
```

3. Suppose you have hidden away 50 dollars to be used for some future emergency. Assuming an inflation rate of 12 per cent per year, write a program to compute how much money, to the nearest dollar, you would need at the end of each of the next 15 years to be equivalent to the buying power of 50 dollars at the time you hid it.

4. Trace the following program. That is, give the values of the variables together with any output after the execution of each statement. Use as data entries 95 110 85 -1.

```
010   REM PROGRAM CLASS
020   SUM=0
030   PRINT "ENTER MARKS ONE TO A LINE,-1 AT END"
040   INPUT MARK
050   NUMBER=0
060   REM REPEAT UNTIL MARK=-1
070   IF MARK=-1 THEN GOTO 190
080      REM IF...THEN...ELSE
090      IF(MARK>0)AND(MARK<=100) THEN GOTO 130
100         REM DO ELSE
110         PRINT"**ERROR:MARK=";MARK
120         GOTO 160
130         REM DO THEN
140         SUM=SUM+MARK
150         NUMBER=NUMBER+1
160      REM END OF SELECTION
170      INPUT MARK
180      GOTO 070
190   REM END OF LOOP
200   PRINT "AVERAGE=";SUM/NUMBER
210   END
```

5. Write a program that reads the following data entries and calculates the average of (a) each of the two columns of data, and (b) each row of the data. You should either precede the data with a number giving the count of the following data items or add a dummy entry following these data items.

92	88
75	62
81	75
80	80

55 60
64 60
81 80

6. Write a program which reads in a sequence of grades (0 to 100) and outputs the average grade (rounded to the nearest whole number), the number of grades and the number of failing grades (failing is less than 50). Assume that a dummy grade of 999 will follow the last grade. See that your output is clearly labelled. Answer the following questions:

(a) What will happen if the grade 74 is misentered as 7 4?

(b) What will happen if the dummy grade 999 is left off? (You can try this.)

(c) What will your program do if there are no grades, that is, if 999 is the only data item?

(d) What will happen if the two grades 62 and 93 are misentered as 6293?

Test your program using the following data:

85 74 44 62 93
41 69 73 999

7. Write a program which determines the unit price (cents per ounce) of different boxes of laundry soap. Round the unit price to the nearest cent. Each box will be described by an entry of the form:

pounds ounces price

5 0 125

This box of soap has a rounded unit price of 2 cents per ounce. Make up about 10 data entries describing soap boxes; if you like, use real examples from a supermarket. You are to precede these entries with one data entry containing a single integer giving the number of soap box entries. Do not use a dummy entry to mark the end of the data. Output a nicely labelled table giving weights, prices in cents, and unit costs. Answer the following questions:

(a) What would your program do if the above example data entry were mistyped as

50 125

(b) Would it be possible to make your program "smart enough" to detect some kinds of improper data? How or why not?

8. Try using the TRACE command on some of your programs that have loops.

9. Rewrite the program VOTING given in this chapter to use a CASE construct.

Chapter 7

STRUCTURING CONTROL FLOW

In the last chapter we introduced the two kinds of constructs that cause an alteration from the linear flow of control in a program. One kind caused looping, the counted FOR or the REPEAT UNTIL; the other caused branching, or selective execution, the IF...THEN...ELSE. Other than the counted FOR loop we had to construct these out of the Basic statements IF...THEN and GOTO. Learning to handle these two kinds of constructs is absolutely essential to programming. And learning to handle them in a systematic way is essential to structured programming.

STRUCTURE OF LOOPS

It is hard to appreciate, when you first learn a concept like loops, that all loops are basically the same. They consist of a sequence of statements in the program that:

1. Initialize the values of certain variables that are to be used in the loop. These consist of assigning values to
   (a) variables that appear in the condition of a REPEAT UNTIL conditional loop.
   (b) variables that appear in the body of the loop on the right-hand side of assignment statements.
2. Indicate that a loop is to commence. If it is a REPEAT UNTIL give the condition that is to control the number of repetitions. If it is a counted FOR loop give the number of repetitions. Each of the two types of loops has a control phrase that determines the number of repetitions.
   (a) A counted loop's repetition is controlled by the control phrase after the FOR, for example

```
FOR I=1 TO 20
```

(b) In the REPEAT UNTIL conditional loop, the control phrase is after the IF, for example

IF I > 20

The condition should contain at least one variable.

3. Give the list of statements, called the body of the loop, that are to be executed each time the loop is repeated. If the loop is a conditional loop, then within the body of the loop there must be some statements that assign new values to the variables appearing in the condition. Usually there is only one variable and its value may be changed by either

   (a) an assignment statement or

   (b) an INPUT statement.

4. At the end of the loop, control is returned to the beginning. In a FOR loop this is done by NEXT; in a REPEAT UNTIL it is done by a GOTO.

5. Give the next statement to be executed once the looping has been carried out the required numbers of times. Control goes from the beginning of the loop to this statement when

   (a) the condition of the REPEAT UNTIL is true or

   (b) the value of the index controlling the counted FOR loop has already reached the value indicated after the word TO.

It is to be noted very carefully that there is no *exit* from the loop, except from one fixed location, and this exit *always* goes to the statement immediately after the end of the loop. It is never possible to go somewhere else in a program. In this way we keep track of control flow and never have the possibility of getting confused about its path. The Basic GOTO statement for altering the path of control permits you to send control to any statement within your program. Since computer scientists came to recognize the importance of proper structuring in a program, the freedom offered by the GOTO statement has been recognized as not in keeping with the idea of structures in control flow. For this reason we will use it only as part of a REPEAT UNTIL or an IF...THEN...ELSE construct.

For many good programmers it has long become a habit to restrict the use of the GOTO in this way. They might also use it to leave the body of a loop somewhere in the middle and exit to the statement following the end of the loop. An exit inside the body is usually related to a second condition. This second condition can be incorporated in the condition of the REPEAT UNTIL by having a compound condition. We will look at examples of this later in the chapter.

FLOW CHARTS

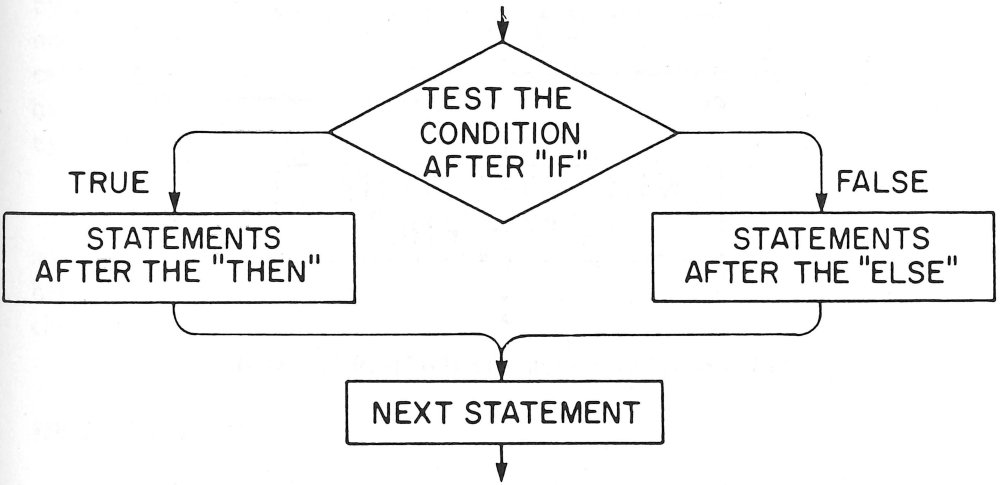
A flow chart is a diagram made up of boxes of various shapes, rectangular, circular, diamond, and so on, connected by lines with directional arrows on the lines. The boxes contain a description of the statements of a program and the directed lines indicate the flow of control among the statements. The main purpose of drawing a flow chart is to exhibit the flow of control clearly, so that it is evident both to the programmer and to a reader who might want to alter the program.

A method of programming that preceded the present method of structured programming found that drawing a flow chart helped in the programming process. It was suggested that a first step in writing any program was to draw a flow chart. It was a way of controlling complexity.

When we limit ourselves to the two standard forms of altering control flow, the loop and the selection constructs, there is little need to draw these flow charts. In a sense, especially if it is properly paragraphed, the program is its own flow chart; it is built of completely standard building blocks.

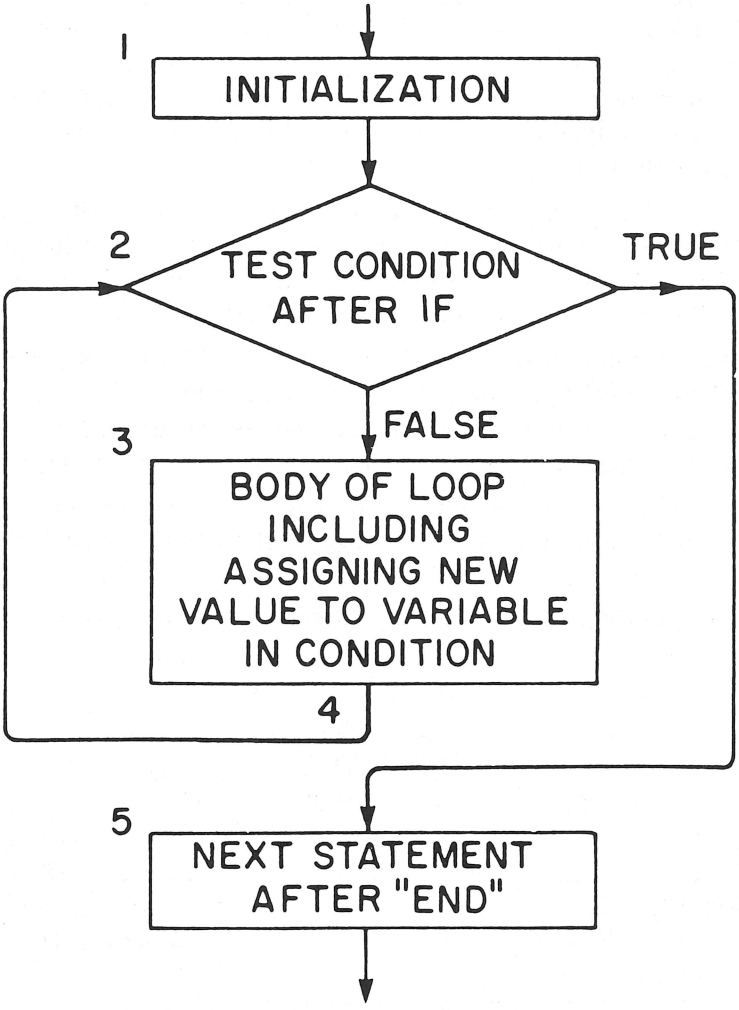
Perhaps it would be helpful to show what the flow charts of our two basic building blocks would be like in case you wanted to draw a flow chart for your whole program.

Here is the flow chart for an IF...THEN...ELSE construct.



FLOW CHART FOR IF...THEN...ELSE

For the REPEAT UNTIL conditional loop that we described, the flow chart would be as shown. The various phases are numbered.



FLOW CHART FOR CONDITIONAL LOOP

The flow chart for the counted FOR would be similar except that box 2 would initialize the index to its first value, increment it by the required amount and then test to see if the loop has been executed the specified number of times.

With these elementary diagrams and the ordinary straight line sequence, flow charts for all Basic programs that we intend to write can be built. In a way, because they are so obviously related to the program, they do not really need to be drawn.

Because structures can be nested one inside the other, any one of the rectangular boxes in these diagrams may be replaced by a sequence of rectangular boxes, or either one of the two basic diagrams themselves. An important thing to notice is that into each of these elementary blocks there is a single entrance and, from each, a single exit. This is critical to maintaining good structure.

PROBLEMS WITH LOOPS

Certain errors are very common with loops. With counted loops the likelihood of errors is much smaller, since the initialization and alteration of the index are done by the FOR itself. You can, however, forget to initialize a variable that is used in the body of the loop. Another problem comes if by chance the index that is used to count the loop is altered in the loop body. The letters I, J, and K are often used as indexes, and you must be careful not to use them again before a previous use is finished. This can happen when one FOR loop is nested inside another and the index I is used by mistake for both loops. You also might write

```
FOR I=1 TO N
```

and forget to initialize N. You should trace the execution of all loops by hand to see if the first iteration is working correctly. Then you should also check the last one.

NESTED LOOPS

We will now look at the more complicated loops. The program sums the marks of students in 4 subjects and outputs these with the average, to the nearest mark. There are a number of lines of entries, five for each student: the student number and the four marks. An entry with the total number of students precedes the mark entries. We will test the program first using only two students to see if it is working.

```
010    REM PROGRAM CLASS
020    REM A SAMPLE PROGRAM WITH NESTED LOOPS
030    INPUT "NUMBER OF STUDENTS=";NUMBEROFSTUDENTS
040    PRINT "INPUT STUDENT NUMBER THEN 4 MARKS"
050    FOR I=1 TO NUMBEROFSTUDENTS
060        SUM=0
070        INPUT STUDENTNUMBER
080        FOR J=1 TO 4
090            INPUT MARK
100            SUM=SUM+MARK
110            NEXT
120        AVERAGE=INT(SUM/4+0.5)
130        PRINT "AVERAGE FOR STUDENTNO ";STUDENTNUMBER;"=";A
140        NEXT
150    END
```

Here is the display:

```
NUMBER OF STUDENTS=2
INPUT STUDENT NUMBER THEN 4 MARKS
? 205
?  55
?  60
?  65
?  70
AVERAGE FOR STUDENTNO  205=63
? 208
?  83
?  81
?  96
?  90
AVERAGE FOR STUDENTNO  208=88
```

You will notice that when we have one FOR loop nested inside another we use two levels of indentation to indicate the control structure.

AN EXAMPLE PROGRAM

We will illustrate some details about loops with another program. Like some of our previous examples, this program outputs a zigzag. However, our new program is smarter than the old ones in that it can output different sizes of and numbers of zigs and zags, depending on the input data.

```
010     REM PROGRAM PICKZIG
020     INPUT "COUNT=";COUNT
030     INPUT "SIZE=";SIZE
040     FOR OUTER=1 TO COUNT
050         PRINT " ****"
060         FOR INNER=1 TO SIZE
070             PRINT " *"
080             NEXT
090         PRINT " ****"
100         FOR INNER=1 TO SIZE
110             PRINT "    *"
115             NEXT
120         NEXT
130     END
```

This program, given the data values 3 and 2 for COUNT and SIZE, outputs the following pattern. We have shown in parentheses values of OUTER and INNER during the output of each of the single-star lines.

```
                                    (OUTER,INNER)
 ****
 *                                      (1,1)
 *                                      (1,2)
 ****
   *                                    (1,1)
   *                                    (1,2)
 ****
 *                                      (2,1)
 *                                      (2,2)
 ****
   *                                    (2,1)
   *                                    (2,2)
 ****
 *                                      (3,1)
 *                                      (3,2)
 ****
   *                                    (3,1)
   *                                    (3,2)
```

The first data value, 3, caused the sub-pattern

```
 ****
 *
 *
 ****
```

```
*
*
```

to be displayed three times. The second data value, 2, was used in determining the height of this sub-pattern.

As you can see in the program, there are two separate loops inside the main FOR loop. Both of these loops use the variable INNER as a counting variable. There is no difficulty using INNER in this way. Each time one of these two loops is entered, INNER is set back to have the value 1.

We can change the pattern displayed by changing the data entries. For example, the following pattern is displayed for data values 1 and 1.

```
                                (OUTER,INNER)
****
*                                 (1,1)
****
  *                               (1,1)
```

In this case, the loops are each executed only once.

Now let us try data values 2 and 0. The pattern is

```
****
*
****
  *
****
*
****
  *
```

In this case the main loop is executed twice, and the inside loops are executed once since in the inner FOR loops the limit is less than the start. We get one execution anyway.

The data values were used by this program to determine the pattern to output. You could write other programs which output different patterns for different data.

LOOPS WITH MULTIPLE CONDITIONS

Sometimes we must terminate a loop if something happens that is unusual. One of the conditions controlling the loop is the standard one; the other is the unusual one. All loops with double-headed conditions must be REPEAT...UNTIL loops, since the counted FOR loop does not allow for the possibility of a second condition. We have seen that compound

conditions can be formed from two or more simple conditions using the AND and OR Boolean operators.

As an example, we will write a program to look for a certain number in a list of numbers that you enter. If you find it in the list, immediately output the position it occupies in the list; if it is not in the list, output NOT IN LIST. The list will be positive integers terminated by -1.

```
010     REM PROGRAM HUNT
020     INPUT "NUMBER YOU ARE LOOKING FOR="; NUMBER
030     PRINT "START TYPING IN LIST;FINISH WITH -1"
040     INPUT NO
050     I=1
060     REM REPEAT UNTIL(NO=NUMBER) OR (NO= -1)
070     IF(NO=NUMBER) OR (NO= -1) GOTO 110
080         I=I+1
090         INPUT NO
100         GOTO 070
110     REM END OF LOOP
120     PRINT
130     REM IF NO=NUMBER THEN...ELSE
140     IF NO=NUMBER THEN GOTO 180
150         REM DO ELSE
160         PRINT "NOT IN LIST"
170         GOTO 200
180         REM DO THEN
190         PRINT "I=";I
200     REM END OF SELECTION
210     END
```

Here is a sample display:

```
NUMBER YOU ARE LOOKING FOR=35
START TYPING IN LIST; FINISH WITH -1
? 12
? 16
? 25
? 35

I=4
```

Notice that in the body of the loop the variable NO in the condition can be changed by the INPUT NO statement; it is initialized outside the loop by the INPUT NO statement in line 040. Since an index I is required to give the position of the number in the list, it must be incremented in the statement I=I+1 and initialized to 1 outside the loop.

IF STATEMENTS WITH MULTIPLE CONDITIONS

Just as the conditional loop can have multiple conditions, so also can IF statements. These can be used very effectively to avoid nesting of IF statements. Suppose you want to count people in a list who fall into a particular age group, say 18-65, as adults. The following program will count the number in the category ADULT in the list. The list of ages is terminated by a -1.

```
010    REM PROGRAM WORKERS
020    DUMMY=-1
030    PRINT "START TYPING LIST OF AGES; FINISH WITH -1"
040    INPUT AGE
050    ADULT=0
060    REM REPEAT UNTIL AGE=DUMMY
070    IF AGE=DUMMY THEN GOTO 110
080       IF (AGE>=18) AND (AGE<=65) THEN ADULT=ADULT+1
090       INPUT AGE
100       GOTO 070
110    REM END OF LOOP
120    PRINT
130    PRINT "NUMBER OF ADULTS=";ADULT
140    END
```

Here is a sample display:

```
START TYPING LIST OF AGES; FINISH WITH -1
? 16
? 25
? 31
? 12
? 28
? 29
? 69
? -1

NUMBER OF ADULTS=4
```

The IF with the compound condition could have been replaced by the more awkward construction with a nested IF statement:

```
IF AGE>=18 THEN
   IF AGE<=65 THEN
      ADULT=ADULT+1
```

but this is not advisable. Notice that since there was no ELSE we could

replace the IF...THEN...ELSE construct by the Basic IF...THEN statement and did not need the usual remarks.

CHAPTER 7 SUMMARY

In this chapter we have taken a closer look at the loop and selection constructs. We discussed flow charts and presented more complex examples of loops and conditions. The following important terms were discussed.

Flow chart - a graphic representation of a program. A flow chart consists of boxes of various shapes interconnected by arrows indicating flow of control. Basic programs can be represented by flow charts.

Exit from a loop - means stopping the execution of a loop. Exit from a REPEAT UNTIL loop occurs at the beginning of the loop when the loop's condition is found to be true.

GOTO statement - transfers control to another part of a program. The GOTO statement is used as part of the construct for a REPEAT UNTIL conditional loop and an IF...THEN...ELSE. Other than for these two uses we do not use GOTOs. Careless use of GOTO statements leads to complex program structures which are difficult to understand and to make correct.

Nested statements - means statements inside statements. For example, FOR loops can be nested inside FOR loops.

Multiple conditions - conditions which use the AND and OR Boolean operators.

CHAPTER 7 EXERCISES

We will base all the exercises for this chapter on the same problem, which we now describe. A meteorologist keeps records of the weather for each month as a table. The first line of the table gives the number of days of the month.

The following lines give the rainfall, low temperature, high temperature, and pollution count for each day of the month. For example, the data for a month might look this way on the screen

```
LENGTH OF MONTH=31
RAIN          LOW   HIGH     POLLUTION
? 0,          33,   37,      3
?1.2,         34,   39,      3
? 0,          35,   40,      0.5
? 0,          34,   38,      2
              etc.
```

Each of the following exercises requires writing a program which reads one month's weather and answers some questions about the month's weather. To make things easier for you, answers for the first two exercises are given.

1. Find the first rainy day of the month. (The following program finds the required day and is a solution for this exercise.)

```
010 REM PROGRAM WETDAY
020 REM FIND FIRST RAINY DAY
030 INPUT "LENGTH OF MONTH=";MNTHLNGTH
040 PRINT "RAIN";TAB(6);"LOW";TAB(10);"HIGH";TAB(16);"POLLUTION"
050 DAY=0
060 RAIN=0
070 REM REPEAT UNTIL (RAIN<>0) OR (DAY>MNTHLNGTH)
080 IF(RAIN<>0) OR (DAY>MNTHLNGTH) THEN GOTO 120
090     INPUT RAIN,LOW,HIGH,POLLUSHUN
100     DAY=DAY+1
110     GOTO 080
120 REM END OF LOOP
130 REM IF RAIN>0 THEN...ELSE
140 IF RAIN>0 THEN GOTO 180
150     REM DO ELSE
160     PRINT "NO RAINYDAYS"
170     GOTO 200
180     REM DO THEN
190     PRINT "DAY ";DAY;" WAS RAINY"
200 REM END OF SELECTION
210 END
```

Notice that we got into trouble choosing variable identifiers for this program.

2. See if the data entries for the days of the month are reasonable. Verify that the rainfall does not exceed 100 and is not negative. Verify that the temperature lies between -100 and 200 and that the high is at least as large as the low. Verify that the pollution count is neither above 25 nor below zero. (The following program validates the month's data and is a solution for this exercise.)

```
010    REM PROGRAM VERIFY
020    INPUT "LENGTH OF MONTH"; MNTHLNGTH
030    PRINT "RAIN";TAB(6);"LOW";TAB(10);"HIGH";TAB(16);"POLLUTION"
040    FOR DAY=1 TO MNTHLNGTH
050        INPUT RAIN,LOW,HIGH,POLLUSHUN
060        IF(RAIN<0)OR(RAIN>100)THEN
               PRINT " DAY ";DAY;"  WRONG RAIN:";RAIN
070        IF(LOW< -100)OR(LOW>HIGH)OR(HIGH>200)THEN
               PRINT " DAY ";DAY;
                 " WRONG TEMPERATURES:";LOW;SPEC(2);HIGH
080        IF(POLLUSHUN<0)OR(POLLUSHUN>25)THEN
               PRINT " DAY ";DAY;
                 " HAS WRONG POLLUTION:";POLLUSHUN
090        NEXT
100    END
```

3. What was the warmest day of the month, based on the high?

4. What was the first rainy day having a high temperature above 38?

5. What were the days of the month with more than a 5-degree difference between the high and low temperatures?

6. What were the two warmest days of the month?

7. Did the pollution count ever exceed 5 on a day when the temperature stayed above 35?

8. What three consecutive days had the most total rainfall?

9. Was it true that every rainless day following a rainy day had a lower pollution count than the rainy day?

10. Using the first 10 days' data, "predict" the weather for the 11th day. Compare (either by hand or within the program) the prediction with the data for the 11th day.

Chapter 8

ARRAYS

So far in our programming, each memory location for data had its own special name; each variable had a unique identifier. In this chapter we will introduce the idea that groups of data will share a common name and be differentiated from each other by numbering each one uniquely.

DECLARATION OF ARRAYS

Suppose, for example, that we had a list of 50 integers and we wanted the sum of all the numbers. Rather than giving each one of 50 memory locations holding the list a different name, we give the list, or array, a name, say NUMBER, and distinguish the various members of the array by giving each a unique *index*. The index is most often an integer. The first element in the array is called NUMBER(1), the second NUMBER(2) and so on up to NUMBER(50). In order that the Basic interpreter can reserve memory space for an array of variables we must declare that we intend to use an array. We must give the size or *dimension* of the array. This is done in a DIM statement. To declare such an array of variables we use the declaration

```
DIM NUMBER(50)
```

In the declaration, after the name of the array we give, in parentheses, the number of elements in the array namely 50. (Actually, in Basic, there is an element with index 0 so that there are 51 elements in this array. Most of the time we will just ignore this element.) Here is the program that reads 50 integers entered one to a line into the array, adds them up, and outputs the sum:

```
010     REM PROGRAM SUMLIST
020     DIM NUMBER(50)
030     PRINT "TYPE IN 50 INTEGERS"
040     FOR I=1 TO 50
050         INPUT NUMBER(I)
060         NEXT
070     REM ADD NUMBERS
```

```
080     SUM=0
090     FOR I= 1 TO 50
100         SUM=SUM+NUMBER(I)
110         NEXT
120     PRINT "SUM=";SUM
130     END
```

The declaration of the array NUMBER in line 020 indicates that the index has a range from 1 to 50 (actually 0 to 50 if you remember). In the FOR loops, reference to NUMBER(I) refers in turn to NUMBER(1), NUMBER(2), and so on up to NUMBER(50). This is what makes the array such a powerful programming tool we need not refer to each element separately in the program but only to the general element NUMBER(I).

In this example, it is not necessary to read all the numbers and then add them up; we did it that way just to show what is necessary for reading or summing a list. We could have written only one loop, combining the two operations.

```
010     REM PROGRAM READANDSUM
020     DIM NUMBER(50)
025     PRINT "TYPE IN 50 INTEGERS"
030     SUM=0
040     FOR I=1 TO 50
050         INPUT NUMBER(I)
060         SUM=SUM+NUMBER(I)
070         NEXT
110     PRINT "SUM=";SUM
120     END
```

If reading and adding the numbers is all that is required we could do it without an array at all as in this program:

```
010     REM PROGRAM ADDUP
020     PRINT "TYPE IN 50 INTEGERS"
030     SUM=0
040     FOR I=1 TO 50
050         INPUT NUMBER
060         SUM=SUM+NUMBER
070         NEXT
080     PRINT SUM
090     END
```

We really should not use an array if we do not need one.

Usually, there is more to be done that requires having the list still present. For instance we could think of dividing each member of a list by

the sum and multiplying by 100 before we wrote them out. This would express each entry as a percentage of the group. To do this we would add these statements to our original SUMLIST program:

```
111    FOR I=1 TO 50
112        NUMBER(I)=INT(NUMBER(I)*100/SUM+0.5)
113        PRINT NUMBER(I)
114        NEXT
```

In this FOR loop the statements in lines 112 and 113 with the index I result in each member of the list being operated on and changed to a percentage, then displayed.

TWO-DIMENSIONAL ARRAYS

It is possible to have arrays that correspond to entries in a *table* rather than just a single list. For instance, a table of distances between 4 cities might be

	1	2	3	4
1	0	20	30	46
2	20	0	12	20
3	30	12	0	15
4	46	20	15	0

We could call this array DISTNCE. DISTNCE(1,4) is 46 and DISTNCE(3,4) is 15. The first number in the parentheses refers to the row in the table, the second to the column. You can see that DISTNCE(3,1) has the same value as DISTNCE(1,3); the table is symmetric, in this case, about the diagonal line running from top left to bottom right. All entries on this diagonal are zero; the distance from a city to itself is zero.

We must learn how to declare such a two-dimensional array. All that is necessary is to write

```
DIM DISTNCE(4,4)
```

Here, in parentheses after the name of the array, we have the ranges of two indexes separated by a comma. The first index is the number of rows, the second the number of columns. As an example (a useless one as it stands), we will read in this table and store it in the memory. On each input line we will type one element of the data but will type the elements in the first row, then second, etc.:

```
010    REM PROGRAM TABLE
020    DIM DISTNCE(4,4)
030    PRINT "TYPE IN 4 BY 4 TABLE"
040    PRINT "TYPE ENTRIES ONE TO A LINE, BY ROWS"
050    FOR I=1 TO 4
060        FOR J=1 TO 4
070            INPUT DISTNCE(I,J)
080            NEXT
090        NEXT
100    END
```

In this program there is one FOR loop nested inside another. The first NEXT in line 080 goes with the inner loop which uses index J. The second NEXT, in line 090 goes with the outer loop which uses index I. We could write these two lines as

```
080            NEXT J
090        NEXT I
```

but this is not necessary. The indentation of the lines makes it clear which NEXT belongs to which loop. The computer knows too because loops must be properly nested (not interlocking).

We have used I to give the row number, J to give the column number. When I=1 the inner loop has J go from 1 to 4. This means the elements of the array entered first are stored in these variables:

DISTNCE(1,1) DISTNCE(1,2) DISTNCE(1,3) DISTNCE(1,4)

These are the elements in row 1 of the table. In giving the name of each element of a two-dimensional array you write, in parentheses separated by a comma, the values of the two indexes. It is just customary to think of a table in such a way that the first index is the row number, the second the column number. Since our table is symmetric it does not matter if we interchange rows and columns, because we get exactly the same result. For most tables it *does* matter, and you must be careful. A table like this is called a *matrix* by mathematicians.

AN EXAMPLE PROGRAM

We will illustrate the use of two-dimensional arrays in terms of a set of data collected by a consumers' group. This group has been alarmed about the recent rapid rise in price of processed wallalumps. They sampled grocery store prices of processed wallalumps on a monthly basis throughout 1980, 1981, and 1982 and observed that prices varied from 75 cents to 155

cents as the following table shows:

MONTH

	1	2	3	4	5	6	7	8	9	10	11	12
1980	87	89	89	89	85	85	85	75	90	100	100	100
1981	95	95	95	95	90	90	85	90	100	110	120	110
1982	110	110	115	115	115	100	100	110	120	140	145	155

These 36 prices were made available and a program was needed to analyze the price changes.

The following program reads in the data and determines the average price for 1981 (the second of the three years):

```
010    REM PROGRAM COST
020    DIM PRICE(3,12)
030    PRINT "TYPE IN TABLE ROW BY ROW, ONE ENTRY TO A LINE"
040    FOR YEAR=1 TO 3
050        FOR MNTH=1 TO 12
060            INPUT PRICE(YEAR,MNTH)
070            NEXT
080        NEXT
090    REM DETERMINE THE AVERAGE PRICE IN 1981
100    SUM=0
110    FOR MNTH=1 TO 12
120        SUM=SUM+PRICE(2,MNTH)
130        NEXT
140    PRINT " AVERAGE 1981 PRICE: ", INT(SUM/12+0.5)
150    REM ADD STATEMENTS TO CALCULATE OTHER AVERAGES
160    END
```

In this program, the array PRICE is declared so it can have a first index which can range from 1 to 3 and a second index which can range from 1 to 12. Effectively, the PRICE array is a table in which entries can be looked up by month and year. The first index in this example is the row number, the second the column number. The first part of the program uses the data to fill up the PRICE array. The second part of the program sums up the prices for each month during 1981 and calculates the average 1981 price.

We could as well calculate the average price for a particular month. For example, the following calculates the average price in February (the second month):

```
SUM=0
FOR YEAR=1 TO 3
   SUM=SUM+PRICE(YEAR,2)
   NEXT
PRINT " AVERAGE FEB. PRICE: ",INT(SUM/3+0.5)
```

We could calculate the average price for the entire three-year period as follows.

```
SUM=0
FOR YEAR=1 TO 3
     FOR MNTH=1 TO 12
          SUM=SUM+PRICE(YEAR,MNTH)
          NEXT
     NEXT
PRINT " OVERALL AVERAGE: ",INT(SUM/36+0.5)
```

This example has illustrated the use of two-dimensional arrays. It is possible to use arrays with three and more dimensions. For example, our consumers' group might want to record prices for five grades of processed wallalumps (that makes one dimension), for three years (that makes two dimensions), for each month (that makes three dimensions). The array declaration

```
DIM PRICE(5,3,12)
```

would set up tables to hold all this data.

ARRAYS AS DATA STRUCTURES

We have spoken of structured programming and shown how control flow is structured in a program. Now we can speak of the structure of data. Giving variables identifiers that are meaningful has been the only way we could systematize data so far. But with arrays we find that data can be structured or organized into one-dimensional forms called lists, or two-dimensional forms called tables. We can use even higher-dimensional arrays when we need them.

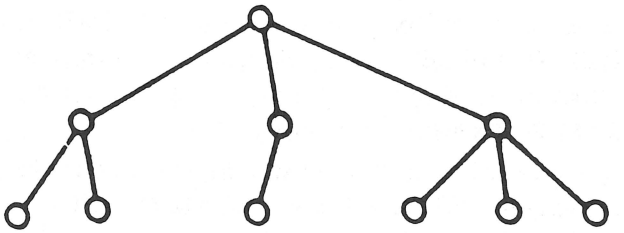
When we approach a problem and want to solve it by creating a computer program we must decide on the data structures we will use. We must decide in particular whether or not we need to establish arrays for any of the data, or whether single variables will serve us well enough.

Arrays will be useful whenever we must store groups of similar pieces of information. They are not necessary when small amounts of information come in, are processed, and then go out.

OTHER DATA STRUCTURES

Just so that you do not think that single variables and arrays are the only kind of data structures we can have, we will mention a few others.

One common structure is the *tree* structure. The easiest way to think of a tree is to imagine a family tree. At the risk of being called chauvinists we will show only the male members in the tree and talk of fathers and sons. This keeps it simple.



TREE STRUCTURE

The diagram shows a man with three sons. The first son has two sons, the second one son, the third three. The grandfather is the *root* of the tree. The tree is of course growing upside down. The lines joining the relatives are called *branches*; the people themselves are *nodes*.

The data we might store could be the names of the people, and the tree structure would have to be stored also. The way it is done is to have *links* or *pointers* stored with the data to give the structure. Each father entry requires a link to each of his sons.

A list can also be arranged with elements and links instead of in an array. This means that some of the information stored is used to describe the data structure and some to give the data. With arrays, the structure is given by the fact that one element follows right next to the preceding element. It does not need a link.

Later we will be investigating other data structures in detail. Often we will use the array structures to implement structures like trees or lists with links from one element to another.

CHAPTER 8 SUMMARY

This chapter has introduced array variables, which are used for manipulating quantities of similar data. An array is made up of a number of elements, each of which acts as a simple, non-array variable. The following terms are used in describing arrays and their uses.

Array declaration - sets aside memory space for an array. For example, the declaration

```
DIM PRICE(4)
```

sets aside space for the array elements PRICE(0), PRICE(1), PRICE(2), PRICE(3), and PRICE(4). Each of these elements can be used like a simple, non-array numerical variable. We often do not use the element with index zero.

Array index (sometimes called array subscript) - used to designate a particular element of an array. For example, in PRICE(I), the variable I is an array index. An array index can be any arithmetic expression that has an integer value.

Array bounds - the range over which array indexes may vary. For example, given the declaration

```
DIM PRICE(3,12)
```

an array element of PRICE can be specified by PRICE(Y,M), where M can range from 0 to 12 and Y can range from 0 to 3.

Out-of-bounds index - an array index which is outside the bounds specified in the array's declaration. This is an error.

Multiply-dimensioned arrays - arrays requiring more than one index, such as the PRICE array given above.

CHAPTER 8 EXERCISES

1. Write a program that will read in the length of a one-dimensional array then read the array itself. It is then to replace each element of the array by the sum of all elements up to and including that element and then output the resulting array.
2. Set the values of the elements of a two-dimensional integer array that has the same number of rows as columns so that the diagonal elements are all +1 and the off-diagonal elements -1. Read in an integer giving the size of a subset of the array and output the subset array one row to a line. You should limit the size to a maximum of 8x8.

3. In mathematics, two one-dimensional arrays of equal length may be multiplied together. The product (sometimes called the scalar product of two vectors) is the sum of the products of corresponding elements in the two arrays. If one array is named A and the other B then the scalar product is the sum, for I going from 1 to the length of the array, of terms of the form

   A(I)*B(I)

   Write a program to find the scalar product of two one-dimensional arrays. Read in the length of the arrays as a variable.

4. In mathematics, two matrices may be multiplied together if the number of columns of the first matrix is equal to the number of rows of the second. The product is a matrix. If we have a matrix A with L rows and M columns and a matrix B with M rows and N columns the elements of the product matrix C are given by the relation

   C(I,J) = sum for K going from 1 to M of A(I,K)*B(K,J)

   Write a program that reads values of L,M,N then reads in matrices A and B computes matrix C and outputs it out. Limit the values of L,M, and N to be 8 or less and use two-dimensional integer arrays to represent the matrices.

Chapter 9

ALPHABETIC INFORMATION HANDLING

We have said that computers can handle alphabetic information as well as perform numerical calculations. But most of the emphasis so far, except for labelling our tables of numerical output, has had very little to do with alphabetic data handling. It is true that we have been dealing with words, like identifiers, but these have been in the Basic programs rather than being handled by them as data. We have, in fact, never had anything but numbers, either real or integer, entered on the keyboard during the execution of a program. In this chapter we will learn how to read in alphabetic data from the keyboard, how to move it from one place to another in the memory of the computer, how to join different pieces of information together, and how to separate out a part of a larger piece of information.

CHARACTER STRINGS

The facilities for handling alphabetic information in standard Basic are somewhat limited. A number of extensions are included in APPLESOFT II Basic. The term "alphabetic information" that we used in the last section is really not general enough to describe what we will learn to handle in Basic. We will be able to handle what you normally mean by alphabetic information, things like people's names -

SARAH MARIE WOOD

but we also want to handle things like street addresses. For example, an address like

2156 CYPRESS AVENUE

includes digits as well as letters of the alphabet. This kind of information we call *alphanumeric* or *alphameric* for short. But that is not all; we want to handle any kind of English text with words, numbers, and punctuation marks, like commas, semicolons, and question marks.

THIS TEXT CONTAINS 7 WORDS; DOESN'T IT?

We have defined a word as being a string of one or more characters preceded and followed by a blank or a punctuation mark, other than the apostrophe. This definition makes 7 a word.

The information we want to handle is any string of characters that may be letters, digits, punctuations marks, or blanks. We tend to think of a blank as being not a character, but a string of blanks is quite different from a string with no characters at all. We call the special string with no characters at all a *null string.* We often write b for the blank character so that you can count how many blank characters are in a string.

HEREbISbAbCHARACTERbSTRINGbSHOWINGbTHEbBLANKS

In Chapter 3 we introduced the characters in the Basic language. In that listing there are more than we have referred to so far in this chapter. The list of special characters includes symbols we need for arithmetic operations +, -, /, *, ^, as well as for making comparisons in logical conditions >, <, =. Then often we used parentheses, not to mention the characters like ; : and , .

One reason we want to be able to handle strings of any of these characters is to be able to work with Basic programs themselves as data. This is the kind of job an interpreter must do, and a programming language like Basic should be suitable for writing an interpreter program. As far as we know interpreters are not, however, written in Basic.

READING AND OUTPUTTING CHARACTERS

We can use variables that hold strings of characters in much the same way as we have been using variables to hold numbers. If we want a variable to store strings of characters, it must be given an identifier that ends with the character $. For example the variable NAME$ is suitable. As before, only the first two characters of the identifier really are used by the Basic interpreter but the whole identifier must not be, or contain, a keyword. We can store a string of any length from zero up to 255 characters in such a variable. If we had these two statements in a program:

```
NAME$="PIERRE"
PRINT NAME$
```

then the computer would display PIERRE in the first field. The string of characters would not have the quotation marks around them. Here is a program that outputs HI.

```
010    REM PROGRAM FRIENDLY
020    CHAR$="H"
030    PRINT CHAR$;
040    CHAR$="I"
050    PRINT CHAR$
060    END
```

This is a rather clumsy way to do the same thing that is done by the statement PRINT "HI". Notice that the PRINT statement in line 030 ended with a semicolon, this is what causes the next PRINT statement in line 040 to continue in the same line without spaces between what is output.

We can also read in the alphanumeric data and output it. The string to be read in must be enclosed in quotation marks. (It is possible to read strings not enclosed in quotation marks provided they do not contain a comma or a colon. If you want to enter several strings on one line they are separated, as numbers are, by commas. That is why the strings cannot contain commas if they have no quotes around them. If you do have a comma it means that one string has ended and another begun. We will begin using quotes around strings but later on, we will get braver.)

Here is a program that reads strings until it finds the word FISH then it outputs STOP TYPING.

```
010    REM PROGRAM CATCH
020    NAME$="BLANK"
030    REM REPEAT UNTIL NAME$="FISH"
040    IF NAME$="FISH" THEN GOTO 070
050        INPUT NAME$
060        GOTO 040
070    REM END OF LOOP
080    PRINT
090    PRINT "STOP TYPING"
100    END
```

Here is a sample display:

```
? "DUCK"
? "GOOSE"
? "FISH"

STOP TYPING
```

There is in Basic a function that will let you find out the length of a string. Its name is LEN. For example, if you type the instruction

```
PRINT LEN("HOW DO YOU DO?")
```

the number 14 will appear on the screen. This is the number of characters in the string. Remember blank is a character.

Here is a program that uses the LEN function:

```
010    REM PROGRAM WORDS
020    PRINT "TYPE A WORD"
030    INPUT WURD$
040    PRINT "THIS WORD HAS ";LEN(WURD$);" LETTERS"
050    END
```

Here is a display for this program:

```
TYPE A WORD
? SICK
THIS WORD HAS 4 LETTERS
```

We had to use a funny variable name WURD because WORD contains the Basic reserved word OR and a syntax error would result. We can however have program names such as WORDS that contain Basic reserved words.

Here is a program to read a list of names and store them in an array of alphanumeric variables. The list will be terminated by a dummy name "ZZZ". We will allow for a maximum length of list of 25 names. We will output the number of names in the list and display the list in reverse order.

```
010    REM PROGRAM NAMES
020    DIM NAME$(25)
030    PRINT "INPUT NAMES ONE TO A LINE"
040    NAME$(0)="NOT ZZZ"
050    I=0
060    REM REPEAT UNTIL NAME$(I)="ZZZ"
070    IF NAME$(I)="ZZZ" THEN GOTO 110
080        I=I+1
090        INPUT NAME$(I)
100        GOTO 070
110    REM END OF LOOP
220    PRINT "NUMBER OF NAMES=";I-1
230    FOR J=I-1 TO 1 STEP -1
135        PRINT NAME$(J)
140        NEXT
145    END
```

Here is a sample display:

```
INPUT NAMES ONE TO A LINE
? MOIRA WHALON
```

```
? DESMOND NEILL
? COLIN FRIESEN
? NORBERT IWANSKI
? ZZZ
NUMBER OF NAMES=4
NORBERT IWANSKI
COLIN FRIESEN
DESMOND NEILL
MOIRA WHALON
```

Here we did not use quotes around the alphabetic information because it did not contain any commas.

SEQUENCING STRINGS

String comparison can be used to sequence strings, to put them in order. Usually we speak of alphabetic order for alphabetic strings.

ABCDEFGHIJKLMNOPQRSTUVWXYZ

The alphabet and digits have the normal order among themselves: 0 comes before 9, A comes before Z. The operators > and < are used to compare the strings. If the strings being compared are not of the same length the longer string is padded on the right with blanks for the purpose of comparison. The following program reads in names and outputs the one that is the last alphabetically. To do this we must be able to compare each string that is read in with the string that is presently the last alphabetically and replace the current final name if the latest one is greater alphabetically.

The names to be examined are entered one to a line starting at the beginning of the line. The end is marked by the dummy name ZZZ.

```
010    REM PROGRAM LAST
020    FURST$="AAA"
030    FINAL$=FURST$
035    PRINT "INPUT NAMES ONE TO A LINE"
040    INPUT NAME$
050    REM REPEAT UNTIL NAME$="ZZZ"
060    IF NAME$="ZZZ" THEN GOTO 100
070        IF NAME$>FINAL$ THEN FINAL$=NAME$
080        INPUT NAME$
090        GOTO 060
100    REM END OF LOOP
110    PRINT "ALPHABETICALLY LAST IS:";FINAL$
120    END
```

Here is the display

```
INPUT NAMES ONE TO A LINE
? GALLER
? TSICHRITZIS
? WORTMAN
? GOTLIEB
? ZZZ
ALPHABETICALLY LAST IS: WORTMAN
```

You will notice that the strings being compared by NAME$>FINAL$ need not be of the same length. For the comparison the result is the same as it would be if the shorter of the two strings were padded with blanks.

JOINING STRINGS TOGETHER

APPLESOFT II Basic strings can be read in, stored, and output. But we can do more than that. We can perform certain operations on them.

The first of these operations is to join two strings together to make one longer string. The operation is called *concatenation* and to do this we use the plus sign. Here is an example of concatenation.

```
010     REM PROGRAM JOIN
020     PRINT "TYPE IN TWO WORDS"
030     INPUT W1$
040     INPUT W2$
050     PRINT W1$+W2$
060     END
```

Here is a sample display:

```
TYPE IN TWO WORDS
?UP
?STAIRS
UPSTAIRS
```

Notice that each word input is on a separate line. We did not put quotes around the words because they contained no commas. Notice also that the operation of concatenation joins the two strings into one string. If we had asked the program to output LEN(W1$+W2$) it would have displayed 8.

Any number of strings may be concatenated at one time; just list them, separated by plus signs.

Here is a program using concatenation that has a loop in it:

```
010     REM PROGRAM BUILD
020     STARS$=""
030     FOR I=1 TO 20
040         STARS$=STARS$+"*"
050         PRINT STARS$
060         NEXT
070     END
```

Here is the display:

```
*
* *
* * *
* * * *
```

etc. until there is a line of 20 asterisks.

In the program the string variable is initialized to the null string which is written as "". Each time the loop is executed, an asterisk is concatenated on to the right-hand end of the string.

SELECTING PARTS OF STRINGS

It is possible using the Basic built-in function MID$ to select parts of strings. We call a part of a string a *substring.* For example, the immediate execution statement:

```
PRINT MID$("MILLION",2,3)
```

would result in the output

ILL

The substring ILL is copied from the string MILLION by starting at the 2nd character and going for 3 characters. The general form is

```
MID$(string, starting position, length of substring)
```

Here is another example:

```
WURD$="MILLION"
PRINT MID$(WURD$,4,LEN(WURD$)-3)
```

The output for this is LION. The same result here can be obtained by

```
PRINT MID$(WURD$,4)
```

If the third item, in parentheses after MID$ is missing it is assumed that the substring is to run to the end of the string. Any attempt to copy more characters than there are in the string beginning at the starting position should cause an execution error. In this example if you wrote

PRINT MID$(WURD$,4,6)

you would still get LION because there are only 4 characters in WURD$ starting with the 4th character and this is less than 6. Unfortunately Basic does not report an error of this sort.

We will now use the MID$ function in an example. The idea is to split a line of text into words. We will input a line which has no punctuation marks and where words are separated by one blank.

```
010    REM PROGRAM SPLIT
015    PRINT "TYPE IN LINE OF TEXT"
020    INPUT LINE$
025    REM ADD BLANK TO THE END OF LINE
030    LINE$=LINE$+" "
040    REM REPEAT UNTIL LEN(LINE$)=0
050    IF LEN(LINE$)=0 THEN GOTO 150
060        PLACE=1
070        REM REPEAT UNTIL MID$(LINE$,PLACE,1)=" "
080        IF MID$(LINE$,PLACE,1)=" " THEN GOTO 110
090            PLACE=PLACE+1
100            GOTO 080
110        REM END OF LOOP
120        PRINT MID$(LINE$,1,PLACE-1)
130        LINE$=MID$(LINE$,PLACE+1)
140        GOTO 050
150    REM END OF LOOP
160    END
```

Here is a sample display:

```
TYPE IN LINE OF TEXT
? HERE IS THE LINE
HERE
IS
THE
LINE
```

There is no real need to have any other function for substrings than MID$ but Applesoft Basic has two others. Functions of the form

LEFT$(string,number of characters in substring)

return the leftmost characters of string.

RIGHT$(string,number of characters in substring)

returns the rightmost characters of string.

AN EXAMPLE PROGRAM

Sometimes a table has different types of information in different columns of a line. To use a two-dimensional array we must have every element of the same type. Instead we use a number of one-dimensional arrays, one for each column of the table. We will now give an example in which the table is the timetable for teachers in a high school. The timetable has been prepared as a table. Each line has a teacher's name, a period (1 to 6), and a room number. We do not use quotes around the names because they do not contain commas.

The input lines look like the following:

```
TEACHER              PERIOD      ROOM
? MS. WEBER,         1,          216
? MRS. THOMPSON,     6,          214
? MRS. JACOBS,       1,          103
? MS. WEBER,         4,          200
? MRS. REID,         2,          216
  ...
? ZZZ,               0,          0
```

A program is needed to output the timetable in order of periods. First, all teachers with their classrooms for the first period should be output, then all teachers with their classrooms for period 2, and so on up to period 6. The output from the program should begin this way:

```
PERIOD              1
MS. WEBER           216
MRS. JACOBS         103
  ...
```

The following program produces this output:

```
010    REM PROGRAM PERIODS
020    DIM TEACHER$(50),PERIOD(50),ROOM(50))
050    PRINT "TEACHER";TAB(15);"PERIOD";TAB(26);"ROOM"
060    COUNT=0
070    INPUT TEACHER$(1),PERIOD(1),ROOM(1)
080    I=1
090    REM REPEAT UNTIL TEACHER$(I)="ZZZ"
100    IF TEACHER$(I)="ZZZ" THEN GOTO 150
110        I=I+1
120        INPUT TEACHER$(I),PERIOD(I),ROOM(I)
```

```
130         COUNT=COUNT+1
140         GOTO 100
150     REM END OF LOOP
160     REM OUTPUT TIMETABLE BY PERIODS
170     FOR NOW=1 TO 6
180         PRINT "PERIOD";TAB(10);NOW
190         FOR J=1 TO COUNT
200             IF PERIOD(J)=NOW THEN
                    PRINT TEACHER$(J);TAB(15);ROOM(J)
220             NEXT
230         NEXT
240     END
```

The first loop in this program reads in the lines representing the timetable. Note that because the alphabetic information of the teachers' names is followed by numerical information we use three one-dimensional arrays, TEACHER$, PERIOD, and ROOM.

Our program is able to read in a timetable consisting of *at most 50 lines.* If there are more than 50 lines in the timetable, then I will eventually be 51 and this will be used as an index for the TEACHER$, PERIOD, and ROOM arrays. This would be an error, because the declarations specify that 50 is the largest allowed array index. The problem is that the index is *out of bounds* when I exceeds 50. You should try to take care that array indexes in your programs stay within their declared bounds.

CHARACTER CODES

In the next example we will make use of two built-in functions in Basic. These are used to change a character to a number, or a number to a character. The function ASC will give the numerical equivalent of a character in the ASCII character code. (The complete ASCII character code is given in Appendix 6.) In this code the numerical value of the character "A" is 65, "B" is 66 "C" is 67,... "Z" is 90. So that if we write

```
PRINT ASC("D")
```

the number 68 would be output.

The function CHR$ does the opposite. If we write

```
PRINT CHR$(68)
```

the letter D would be output.

We will give an example in which each line of data gives a student's name and a grade of A, B, or C. The program reads the names and grades. It outputs a sentence about each student's work and reports the number of A grades, B grades, and C grades.

```
010    REM PROGRAM GRADES
020    REM INITIALIZE COUNTS OF GRADES TO ZERO
030    DIM COUNT(3)
040    FOR I=1 TO 3
050        COUNT(I)=0
060        NEXT
070    INPUT "CLASS SIZE=";SIZE
080    REM READ IN GRADES,COMMENT, AND TABULATE
090    FOR J=1 TO SIZE
100        REM GRADE LEVELS ARE A, B, AND C
110        INPUT NAME$, LEVEL$
120        PRINT NAME$;
130        IF LEVEL$="A" THEN PRINT " DID EXCELLENT WORK"
140        IF LEVEL$="B" THEN PRINT " DID GOOD WORK"
150        IF LEVEL$="C" THEN PRINT " DID FAIR WORK"
160        INDEX=ASC(LEVEL$)-64
170        COUNT(INDEX)=COUNT(INDEX)+1
180        NEXT
190    PRINT
200    FOR K=1 TO 3
210        PRINT COUNT(K);" STUDENT(S) WITH GRADE ";CHR$(K+64)
220        NEXT
230    END
```

Here is a sample display:

```
CLASS SIZE=3
?R. MARTY,  B
R. MARTY  DID GOOD WORK.
?R. SCHILD,  A
R. SCHILD DID EXCELLENT WORK.
?M. GREEN,  B
M. GREEN  DID GOOD WORK.

1 STUDENT(S) WITH MARK A
2 STUDENT(S) WITH MARK B
0 STUDENT(S) WITH MARK C
```

This example used the functions ASC and CHR$. A similar pair of functions can transform a number, either real or integer, into a string and vice

versa. The value of the function

STR$ (arithmetic expression)

is a string which represents the value of the arithmetic expression. The value of the function

VAL (string expression)

is the numerical value that corresponds to the string expression. At least the first character of the string must be a possible item in a number or VAL has a value zero. For example,

VAL ("A")

is zero.

VAL ("21A")

is 21.

CHAPTER 9 SUMMARY

In this chapter we have given methods of manipulating strings of characters. The following important terms were presented.

Length of a string - The number of characters in a string.

LEN - the built-in Basic function that has a value equal to the length of the string that is in parentheses after LEN; LEN is the name of this function. The form

LEN(string expression)

has an integer value which is the number of characters in the evaluated string expression.

String comparisons - used to test character strings for equality and for ordering. If strings are not of equal length the shorter string is padded on the right with blanks for the comparison.

< comes before (less than)
> comes after (greater than)
<=comes before or is equal (less than or equal)
>=comes after or is equal (greater than or equal)
= equal
<>not equal

Concatenation - means putting strings together, one after the other, to form another string. This is accomplished by using the plus sign. The statement

PRINT "PINE"+"APPLE"

is to output PINEAPPLE. Any number of strings may be concatenated. The general form is

string+string{+string}

where the string may be a string constant or an alphanumeric variable. The curly brackets indicate that there may be any number of strings beyond two in a concatenation.

Substring - means a part of a string.

MID$ - a built-in function whose value is a substring. For example,

PRINT MID$("NAIL",2,3)

would result in the output AIL. In general,

MID$(string,starting position,length of substring)

The length should not exceed what is possible given the string and starting position. If the length of the substring is omitted the substring goes up to the end of the string.

LEFT$ - the substring starting at the leftmost character of the string and having length characters. The form is

LEFT$(string,length of substring)

RIGHT$ - the substring giving the rightmost length characters of string. The form is

RIGHT$(string,length of substring)

ASC - the built-in Basic function used to change a character into its ASCII code equivalent. The expression

ASC(string expression)

has an integer value. The value is the ASCII equivalent of the first character in the value of the string expression. The ASCII equivalents of all characters are listed in Appendix 6. In particular, the letter A has code 65 as a decimal number; the other letters follow successively.

CHR$ - the built-in Basic function whose value is the character corresponding to the decimal value of the function's argument. The argument of a function is in parentheses after the function's name. The form

CHR$(integer expression)

yields the ASCII character corresponding to the value of the expression. The value must be between 0 and 255 or an error message will occur.

STR$ - a built-in Basic function for changing a number into a string.

VAL - a built-in Basic function for changing a string into a number. At least the first part of the string must have a numerical equivalent or zero is the returned value.

CHAPTER 9 EXERCISES

1. Which of the following comparisons of strings are true?

   (a) "DAVID BARNARD" = "DAVID BARNARD"
   (b) "E. WONG " = "EDMUND WONG"
   (c) "MARK FOX " = "MARK FOX"
   (d) "JOHNSTON" > "JOHNSON "
   (e) "416 ELM ST " < "414 ELM STREET"
   (f) "HUME,PAT" >= "HOLT,RIC"
   (g) "ALLEN" <> "ALAN "

2. Read in a phrase P from a single line, such as

   ONE SWALLOW DOESN'T MAKE A SUMMER

or

   AN OUNCE OF PREVENTION IS WORTH A POUND OF CURE

Write statements to accomplish each of the following.

   (a) Find the first blank in P and set its location into the integer variable FIRSTBLANK.

   (b) Set the integer variable LAST to the location of the beginning of the last word in P.

   (c) Change P by adding a period at the end of the phrase.

   (d) Change P by replacing its first word by the character "1".

   (e) Change P by extending it on the right by the phrase THEY SAY.

   (f) Set the integer variable COUNT to the number of words in P. You can assume that each word, except the last word, is followed by a single blank.

3. Write a program which looks up Nancy Wong's telephone number and outputs it. You are given a set of data lines, each containing a name and a phone number. For example, the first line might be

   JOHN ABEL, 443-2162

4. Write a program that checks to see that the "I before E except after C" rule in spelling is followed. Your program should read some text which appears as a series of strings. It should search for I and E appearing next to each other. If the combination is EI and is not immediately preceded by a C, then the string should be output together with an appropriate warning message. Similar action should be taken if C immediately precedes IE.

5. Write a program that will accept names of persons (first and last) in the form

LOUISA MOLYNEUX

and output

MOLYNEUX, L.

Make sure your program will work for names already abbreviated to initials. Arrange that the program will work for a number of names on each run.

6. Write a program which reads text and determines the percentage of words having three letters. For simplicity, use text without any punctuation.

7. What does the following program output?

```
010    REM PROGRAM FLOWERS
020    DIM POEM$(2)
030    POEM$(1)="A ROSE"
040    POEM$(2)=" IS "
050    FOR SAYITAGAIN=1 TO 3
060        FOR PART=1 TO 2
070            PRINT POEM$(PART)
080            NEXT
090        NEXT
100        PRINT POEM$(1)
110    END
```

8. What else does this program output?

```
010    REM PROGRAM POLISH
020    DIM NAME$(20)
030    DIM PRICE(20)
040    INPUT "NUMBER OF BRANDS=";N
050    FOR I=1 TO N
060        INPUT NAME$(I),PRICE(I)
070        NEXT
080    INPUT DEAL
090    FOR I=1 TO N
```

```
100         IF PRICE(I)>DEAL THEN PRINT NAME$(I)
110         NEXT
120     END
```

Here is the display up to a point:

```
NUMBER OF BRANDS=3
JOHNSONS,  518
LEMON OIL, 211
DOMINO,    341
  300
```

9. Write a program which reads yesterday's and today's stock-market selling prices and outputs lists of rapidly rising and rapidly falling stocks. A typical data entry will look like this:

GENERAL ELECTRIC, 93.50, 81.00

The entry gives you the company's name followed by yesterday's price, followed by today's price. Your program should output a list of companies whose stock declined by more than 10 per cent, and then a list of companies whose stock rose by more than 10 per cent.

Chapter 10

STRUCTURING YOUR ATTACK ON THE PROBLEM

STEP-BY-STEP REFINEMENT

Most of the examples of programming so far have been short examples. Nevertheless we have emphasized *some* of the aspects of good programming. These were:

1. Choosing meaningful words as identifiers.
2. Placing remarks in the program to increase the understandability.
3. Paragraphing loops and selection constructs to reveal the structure of control flow and adding remarks at the beginning and end of these constructs.
4. Choosing appropriate data structures.
5. Reading programs and tracing execution by hand, to strive for correctness before machine testing.

All of these are important even in small programs, but it is only when we attempt larger programs that our good habits will really start to pay off.

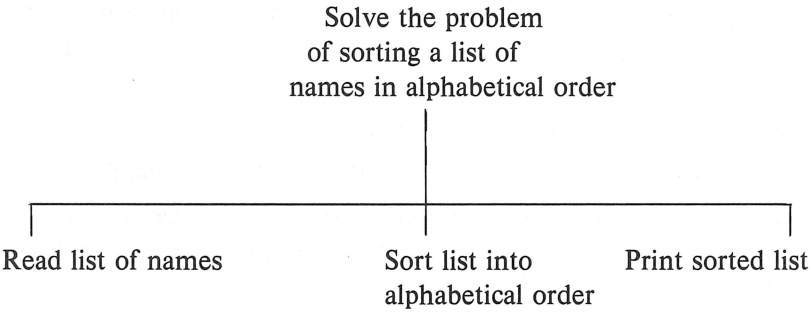
And when we work on larger programs we will find that we have something else to structure, and that is our attack on the problem. To solve a problem we must move from a statement of what the problem to be solved is, to a solution, which is a well-structured program for a computer. The language of our program will be Basic.

The original statement of a problem will be in English, with perhaps some mathematical statements. The solution will be in Basic. What we will look at in this chapter is the way we move from one of these to the other. We will be discussing a method whereby we go step by step from one to the other. This systematic method we will refer to as *step-by-step refinement.* Sometimes we say that we are starting at the top, the English-language statement of the problem, and moving down in steps to the bottom level, which is the Basic program for the solution. We speak of the *top-down approach* to problem solution.

TREE STRUCTURE TO PROBLEM SOLUTION

To illustrate the technique of structuring the solution to a problem by the step-by-step refinement, or the top-down approach, we need a problem as an example. We need a problem that is large or difficult enough to show the technique, but not so large as to be too long to follow. If a program is too long and involved we will use another technique that divides the job into modules and does one module at a time. This is called *modular programming.* It is another form of structured programming. But it must wait until we have learned about subroutines.

The example we choose is sorting a list of names alphabetically. We will now start the solution by trying to form a tree which represents the structure of our attack. The root of the tree is the statement of the problem. In the first move we show how this is divided into three branches:



At each of the three nodes that descend from the root we have an English statement. These statements are still "what-to-do" statements, not "how-to-do-it." A statement of how to do something is called an *algorithm* for doing it. A set of instructions for assembling a hi-fi amplifier is an algorithm for making a hi-fi amplifier. A cake recipe in a cookbook is an algorithm for making a cake. The problem of making a cake is solved by following the recipe.

We will be moving down each branch of the solution tree replacing a statement of "what to do" by an algorithm for doing it. The algorithms will not necessarily be in the Basic language. We will use a mixture of English and Basic at each node until in the nodes farthest from the tree root we have a Basic program.

CHOOSING DATA STRUCTURES

Before we try to add more branches to the solution tree, we should decide on some data structures for the problem of sorting the list of names. We need not make all the decisions at this stage, but we can make a start.

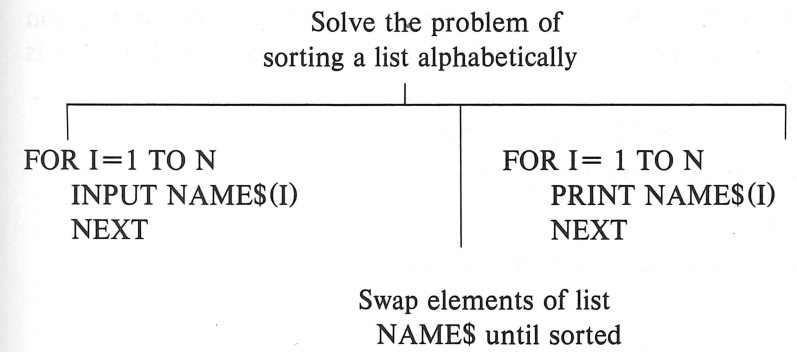
We will use an array of character variables called NAME$ to hold the list of names to be sorted. The length of this list we will call N. We will allow a maximum size for the list of 100 names. This means we must have an array declaration such as

```
DIM NAME$(100)
```

The actual list will have N names, and we must read this number N in as part of the input. We will put one name on each input line, left-justified. For indexing the list we will need an index I. We will assume for the moment that we will keep the sorted list in the same locations as the original list. The names will have to be rearranged, and this means some swapping will be needed. We will use the variable TEMP$ as a temporary variable for swapping.

GROWING THE SOLUTION TREE

Having decided on at least some of the data structures, we are prepared to continue the process of structuring the solution tree. We can see how to develop the left and right branches now, even as far as transforming them into Basic program segments. The middle branch can be refined a little by saying that sorting will be accomplished by element swapping. Here is the tree now:



At this stage we must obviously face up to designing an algorithm for producing a sorted list by swapping.

DEVELOPING AN ALGORITHM

We want the names to be in the sorted list so that each name has a larger value than the name ahead of it in the list.

In the sorted list

HOLT
HORNING
HULL
HUME

we see that each name is alphabetically greater than the one preceding it.

In our solution tree one branch must be developed further; this is, "Swap elements of list NAME$ until sorted." We have seen from the example that a sorted list has the largest value in the last position. This is also true of the list if an element is removed from the end. The new last element is the largest one of the smaller list. So our next refinement in the solution is to arrange the list in this way. We write:

```
FOR LAST varying from N to second,
    swap elements so largest is in LAST
```

Suppose that the list of four names we had was originally in this order:

HORNING
HUME
HOLT
HULL

LAST begins with a value of 4, and the names are swapped so HUME is in position 4. Then LAST is set to 3 and the names are swapped so HULL is in position 3. Then LAST is set to 2 and HORNING is placed in position 2. The only remaining name, HOLT, is left in position 1, and the list is alphabetized.

The first part of this can be written in Basic as

```
FOR LAST=N TO 2 STEP -1
```

But we must still refine the part,

```
Swap elements so largest is in LAST
```

It becomes

```
FOR I varying from first TO (LAST-1)
    IF element(I) > element(I+1) THEN
        swap elements
```

The first two parts of this can now become Basic; this produces

```
FOR I=1 TO LAST-1
    IF NAME$(I) > NAME$(I+1) THEN
        swap elements
    NEXT
```

We must now refine the statement "swap elements". It is

```
TEMP$=NAME$(I)
NAME$(I)=NAME$(I+1)
NAME$(I+1)=TEMP$
```

Because these three statements go after the THEN, we will use the construct for IF...THEN...ELSE rather than a simple IF...THEN.

Now we can assemble the complete program.

THE COMPLETE PROGRAM

```
010    REM PROGRAM SORT
020    REM SORT LIST OF N NAMES ALPHABETICALLY
030    DIM NAME$(100)
040    REM INPUT NAME LIST
050    INPUT "LIST LENGTH=";N
060    PRINT "ENTER NAMES ONE TO A LINE"
070    FOR I=1 TO N
080        INPUT NAME$(I)
090        NEXT
110    REM SWAP ELEMENTS OF LIST UNTIL SORTED
120    REM LOOP WITH LAST VARYING FROM N TO SECOND
130    REM SWAP ELEMENTS SO LARGEST VALUE IS LAST
140    FOR LAST=N TO 2 STEP -1
150        REM WITH I VARYING FROM FIRST TO LAST-1
160        REM IF ELEMENT(I) > ELEMENT(I+1)
170        REM SWAP THESE ELEMENTS
180        FOR I=1 TO LAST-1
185            REM IF NAME$(I)>NAME$(I+1)THEN...ELSE
190            IF NAME$(I) > NAME$(I+1) THEN GOTO 220
200                REM DO ELSE
210                GOTO 260
220                REM DO THEN
230                TEMP$=NAME$(I)
240                NAME$(I)=NAME$(I+1)
250                NAME$(I+1)=TEMP$
260            REM END OF SELECTION
270            NEXT
```

```
275         NEXT
280     REM OUTPUT SORTED NAME LIST
285     PRINT "HERE IS THE SORTED LIST"
290     FOR I=1 TO N
300         PRINT NAME$(I)
310         NEXT
320     END
```

Here is a sample display:

```
LIST LENGTH=5
ENTER NAMES ONE TO A LINE
?ANDREWS
?CAMERON
?ALLEN
?BAKER
?DAWSON
HERE IS THE SORTED LIST
ALLEN
ANDREWS
BAKER
CAMERON
DAWSON
```

Notice that the English parts of the solution tree remain as remarks in the final program. Comments are not added after a program is written, so that it can be understood at a later date, but are an integral part of the program construction process.

ASSESSING EFFICIENCY

In this approach to problem solution we have moved step by step to refine the statement of the problem in English into a program in a language that is acceptable to a computer, namely Basic. In the process, as we constructed the solution tree, we gradually replaced statements of what is to be done by statements of how it is to be done; we devised an algorithm for performing the process. The algorithm was expressed in English, or a mixture of English and Basic. Then finally we had a Basic program.

Nowhere during this process have we spoken about the efficiency of the method that we have chosen, that is, the efficiency of our algorithm. This is because the issue of efficiency complicates the solution. Since in structured programming we are trying to control complexity, we have in this first attempt eliminated efficiency from our considerations.

This means that to now add the refinement of a more efficient algorithm will require us to back up to an earlier point in the solution tree and redo certain portions. In the step-by-step refinement method of problem solution we do not always move from the top down in the solution tree. In that sense, then, the top-down approach is slightly different. In it you would move always in the one direction. In practice this would be impractical, as afterthoughts must always be allowed to improve a method of solution. The only reason to reject afterthoughts is that the work in incorporating them is not justified, considering the gain that would result.

In our particular example you can see that it is possible, at a certain stage, that the list might be sorted and that there is no need to keep on to the bitter end. What we should incorporate is a way of recognizing that the list is sorted so that the mechanical sorting process can stop.

A BETTER ALGORITHM

What we must do is to back up in the solution tree to the point where we had in the middle branch the words, "Swap elements of NAME$ until sorted." We have translated this essentially by the statement, "Swap elements of NAME$ in such a way that at the end of the swapping process the list is sure to be sorted."

We are going to change now to the statement, "Swap elements of NAME$ in such a way that at the end of the swapping process the list would be sorted, and stop either when the list *is* sorted or when the normal end of the swapping process is reached." You can see that we are going to have a loop now with two conditions. The condition of the swapping process being finished is the same as what we have now. What we must add to the loop is the second condition

list is not sorted

But how do we know when the list is sorted? We must devise a method to test whether or not the list is sorted. You will notice that if the inner loop does not swap any names, the list *must* be sorted. We should have a flag called FINISHED that can be set to 1 to indicate that the list is sorted, or 0 to indicate that the list is not sorted.

The outer loop which would have to be a conditional loop would then begin

```
REM REPEAT UNTIL FINISHED=1 OR LAST<2
IF FINISHED=1 OR LAST<2 THEN GOTO...
```

We would have to initialize this loop by having these instructions precede it.

```
FINISHED=0
LAST=N
```

These set the flag FINISHED to 0 so that the loop will begin properly and start the count. Inside the loop we must perform the adjustment in the index LAST by -1. Since we are now using a conditional loop instead of a counted FOR loop, we must do our own counting. This would mean we need the instruction

```
LAST=LAST-1
```

just before the end of the loop. We want FINISHED to be changed to 1 if *no* swapping takes place in the inner FOR loop. This can be accomplished if we set it to 1 just before we enter the inner loop and return it to 0 if any swapping does take place. The altered part of the program is as follows.

```
110    REM SWAP ELEMENTS OF LIST UNTIL
120    REM EITHER SWAPPING PROCESS IS COMPLETED
130    REM OR THE LIST IS SORTED AS INDICATED
131    REM BY THE FLAG "FINISHED" BEING ONE
132    FINISHED=0
133    LAST=N
134    REM REPEAT UNTIL FINISHED=1 OR LAST<2
140    IF FINISHED=1 OR LAST<2 THEN GOTO 277
150       FINISHED=1
160       FOR I=1 TO LAST-1
170          REM IF...THEN...ELSE
180          IF NAME$(I) > NAME$(I+1) THEN GOTO 220
200             REM DO ELSE
210             GOTO 260
220             REM DO THEN
225             FINISHED=0
230             TEMP$=NAME$(I)
240             NAME$(I)=NAME$(I+1)
250             NAME$(I+1)=TEMP$
260          REM END OF SELECTION
270          NEXT
275       LAST=LAST-1
276       GOTO 140
277    REM END OF LOOP
280    REM OUTPUT SORTED NAME LIST
 (as before)
```

BETTER ALGORITHMS

In our example we could see that an improvement in the efficiency of the sorting algorithm could be achieved, and we backed up the solution tree and redid a portion to incorporate the improvement. This was an easier job than trying to think about efficiency in the first place. This is why in the step-by-step refinement method we do not consider efficiency at first. In a way we were lucky that our algorithm could be modified so readily. We might have done the swapping in an entirely different way, in which we would not be able to detect a sorted list by the absence of swapping on any iteration of the process.

To see how this might be, suppose that to sort this list each element were compared with the first element. If it were smaller, the two would be swapped. With the smallest in the first position the list would be shortened by one and the process repeated. The difficulty here is that the fact that no swapping occurs in any round only means that the smallest is already in the first position, not that the list is sorted. We have no way of seeing that the list is sorted unless we compare each list member with its next-door neighbor. And this is what we did in our sorting method. Our method is known as the *bubble sort.*

So our method is more suited to this particular improvement than a method that involves swapping by comparison of each element with one particular element. If we had started this way we would have had to revise completely. To say that efficiency considerations are left until after a first algorithm is programmed produces disadvantages. For many standard processes like sorting, various algorithms have been explored, their efficiencies evaluated, and a best algorithm determined. The method we have developed is certainly not the best that has been devised.

This best, or optimal, algorithm often depends on the problem itself. For instance, one algorithm may be best for short lists, another for long lists. Establishing "the" best method is very difficult and depends on circumstances. Always try to pick a "good" algorithm if you are programming a standard process. At least avoid "bad" algorithms. Very often, programs are already written using good algorithms and you can use them directly in your own program. We can create programs from modules that are already made for us. Then one of the branches of your solution tree is filled by a *prefabricated module.* We need only learn how to hook it up to our own program. We can also create modules of our own. This technique is called *modular programming* and it is an additional way to conquer problem solving, by dividing the problem into parts.

CHAPTER 10 SUMMARY

In previous chapters we concentrated primarily on *learning* a programming language; we have covered variables, loops, character strings, arrays, and so on. In this chapter, the focus has been on *using* a programming language to solve problems.

The method of problem solving which we described is based on the idea of dividing the problem into parts - the divide-and-conquer strategy. Each of these parts in turn is divided into smaller parts. This continues until eventually the solution to the problem has been broken into small parts which can be written in a programming language like Basic. We will review this method of problem solving using the following terms:

Top-down approach to programming. When using a computer to solve a problem, you should start by understanding the problem thoroughly. You start at the "top" by figuring out what your program is supposed to do. Next you split your prospective program into parts, for example, into a reading phase, a computation phase, and a outputting phase. These phases represent the next level in the design of your program. You may continue by defining the data which these phases use for passing information among themselves, and then by writing Basic statements for each of the phases. The Basic statements are the bottom level of your design; they make up a program which should solve your problem. In larger programs, there may be many intermediate levels between the top - understanding the problem completely - and the bottom - a program which solves the problem. (Beware:top-down program design does *not* mean writing REM PROGRAM name at the top of the page, followed by declarations of arrays, followed by statements! The top level in top-down design means gaining an understanding of the problem to be solved, rather than writing the first line of Basic.)

Step-by-step refinement. When you are writing a program, you should start with an overall understanding of the program's purpose. You should proceed step by step toward the writing of this program. These steps should each refine the proposed program into a more detailed method of solving the problem. The last step refines the method to the level where the computer can carry out the required operations. This means that the final refinement results in a program which can be executed by the computer. As you can see, the idea behind top-down programming is step-by-step refinement leading from the problem statement to the final program.

Tree structures to problem solution. In this chapter we have illustrated top-down programming by drawing pictures of trees. The root, or

base, of the tree is labelled by the statement of the problem. Once the problem has been refined into subproblems, we have our tree grow a branch for each subproblem. In turn, each subproblem can be divided, resulting in sub-branches, and so on. When you are actually solving problems, you will probably not actually draw such a tree. However, you may well use the idea behind drawing this tree, namely, step-by-step refinement leading from problem statement to problem solution.

Use of remarks. One of the purposes of remarks, REM..., in a program is to remind us of the structure of the program. Remarks are used to remind us that a particular sequence of Basic statements has been written to solve one particular part of the problem.

CHAPTER 10 EXERCISES

1. You are to have the computer read a list of names and output the names in reverse order. In your top-down approach to writing your program, you first decided your program should have the overall form:

(a) Read in all of the names

(b) Output the names in reverse order

Next, you decided that the names will be passed from part (a) to part (b) via an array declared by

```
DIM NAME$(50)
```

The index of the last valid name read into this array will be passed to part (b) in a variable called SIZE. Making no changes to this overall form, you must now complete the program. Include remarks at the appropriate places to record the purpose of the two parts of your program. Answer the following questions about your completed program.

- Can you think of another way to write part (a) of your program without changing part (b)? How?

- Can you think of another way to write part (b) of your program without changing part (a)? How?

2. The school office wants a list of all A students and a list of all B students. The input line should be of the form:

```
DAVID TILBROOK, A-
```

Each grade is A,B,C,D or F, which may be followed by + or -. The school's programmer has designed the following two possible structures for a program to read the input and output the two required lists.

First program structure: (a) Read names and grades and save all of them in arrays (b) Output names having A grades (c) Output names having B grades

Second program structure: (a) Read names and grades and save only those with As or Bs in arrays (b) Output names having A grades (c) Output names having B grades

Suppose the final program will have room in arrays to save at most 100 students' names. What advantage does the second program structure have over the first one? You do not need to write a program to answer these questions.

3. A company wants to know the percentage of its sales due to each salesman. For each salesman there is an identification number and the dollar value of his sales. The top-down design of a program to output the desired percentages has resulted in this program structure:

(a) Read in salesmen's numbers and sales and add up total sales

(b) Calculate each salesman's percentage of the total sales

(c) Display the salesmens' numbers and percentages.

Parts (a) and (c) have been written in Basic. You are to write part (b) in Basic, add declarations and complete the program. Here is part (a):

```
200    REM INPUT SALESMEN AND SALES, TOTAL SALES
210    SUM=0
220    I=1
230    INPUT MAN(I),SALES(I)
240    REM REPEAT UNTIL MAN(I)=999
250    IF MAN(I)=999 THEN GOTO 290
260        SUM=SUM+SALES(I)
270        I=I+1
280        INPUT MAN(I),SALES(I)
285        GO TO 250
290    REM END OF LOOP
300    N=I-1
```

Here is part (c) written in Basic:

```
500    REM DISPLAY SALESMEN AND PERCENTAGES
510    PRINT "   SALESMAN";"   PERCENT"
520    FOR I=1 TO N
530        PRINT SALESMAN(I);TAB(10);PERCENT(I)
540        NEXT
```

4. A better sorting algorithm than the bubble sort is called the Shell sort, after the man who devised it Donald Shell. Here is the Basic program segment for the Shell sort. It could be used to replace lines 110 to 275 of the program SORT given in this chapter.

```
110    REM SORT LIST BY SHELL SORT METHOD
115    REM SET INITIAL VALUE OF SPACE
120    SPACE=INT(N/2)
125    REM COMPARE AND SWAP ELEMENTS
130    REM SEPARATED BY SPACE
135    REM REPEAT UNTIL SPACE <= 0
140    IF SPACE <= 0 THEN GOTO 245
145       FOR I=SPACE+1 TO N
150          J=I-SPACE
155          REM REPEAT UNTIL J <= 0
160          IF J <= 0 THEN GOTO 225
165             REM IF...THEN...ELSE
170             IF NAME$(J)>NAME$(J+SPACE) THEN GOTO 190
175                REM DO ELSE
180                J=0
185                GOTO 215
190                REM DO THEN
195                TEMP$=NAME$(J)
200                NAME$(J)=NAME$(J+SPACE)
205                NAME$(J+SPACE)=TEMP$
210                J=J-SPACE
215             REM END OF SELECTION
220             GOTO 160
225          REM END OF LOOP
230          NEXT
235       SPACE=INT(SPACE/2)
240       GOTO 140
245    REM END OF LOOP
```

Prepare a program called SORT2 which incorporates this better method of sorting and test it on a list of names. If the list is reasonably long this method is much faster than a bubble sort.

Chapter 11

THE COMPUTER CAN READ ENGLISH

So far you learned how to handle character strings. You learned how to compare strings, either for the purpose of recognizing particular strings or for putting various strings in order. We also joined strings together and formed substrings from strings. But there are more things that you can do with strings. In this chapter we will show how to create the illusion that the computer does things that we normally associate with people; we might say that it is "intelligent". We say it has an artificial intelligence, since it is of course *not* human, and thinking is what humans do. The field of artificial intelligence in computer science concerns itself with getting the computer to perform acts that we think of as the province of humans. Of course, when we see how it is done, we realize it is just a mechanical process. It has to be mechanical or a machine could not do it. But if you do not know how the "trick" is performed, it does seem as if the machine can "think".

The field of artificial intelligence is involved with many different activities of man as imitated by machine, but one of the most interesting is the way that a machine is made to deal with statements made in *natural language.* We call a language, like English, a natural language because it evolved over a period of time through use. A language like Basic is a *formal language.* It has been defined, it is unambiguous and it is really very limited. Trying to get computers to deal with natural language is a major task. We would like to be able to write questions in natural language and have the computer provide answers to our questions from a bank of information. This is a goal in information retrieval systems.

We have not yet got very far along the way towards question-answering systems in natural language, but it is clear there are basic "skills" the computer must have before it can cope with this. One of these skills is the ability to read.

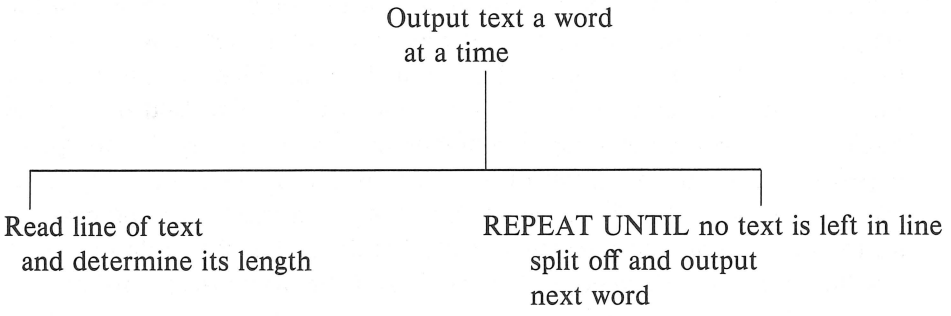
WORD RECOGNITION

When you first learn how to read you must learn to recognize words. To do this you must recognize what a word is. You learn the basic characters, the letters, then you learn that a word is a string of characters with a blank in front and a blank after it and no blanks in between. We are now going to write a program that will input a line of text and split it up into words. To simplify the job, we will begin our problem without any punctuation marks in the text. As an example,

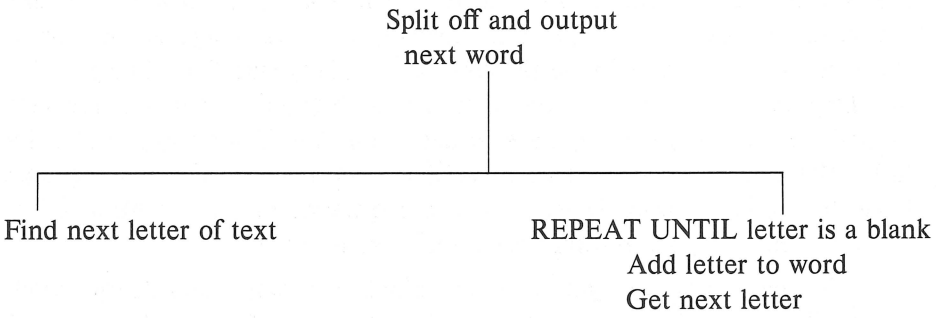
HEREbISbAbTEXTb

where we have used b to represent a blank. The method of dealing with problem solving by simplification is very helpful. Solve a simpler problem before you try a harder one. We will learn to cope with punctuation marks later.

Our solution tree for this problem is:



All the parts are straightforward except "Split off and output next word." We will refine it further:



We will call the line of text we read LINE$; we will repeatedly split off the first letter of LINE$ and store it as SYMBOL$. The first letter will be LEFT$(LINE$,1). We will store the word we find in WURD$. (Note: WORD contains the keyword OR.) As each character is split off it will be stored in SYMBOL$. We can add it to WURD$ by

```
WURD$=WURD$+SYMBOL$
```

We are ready now for the program:

```
010    REM PROGRAM READING
020    PRINT "TYPE LINE OF TEXT, MAX 40 CHAR"
025    INPUT LINE$
030    REM REPEAT UNTIL LINE$="ZZZ"
040    IF LINE$="ZZZ" THEN GOTO 280
070        REM ADD BLANK TO END
080        LINE$=LINE$+" "
090        REM REPEAT UNTIL LINE$=" "
100        IF LINE$=" " THEN GOTO 260
110            REM SPLIT OFF AND OUTPUT NEXT WORD
120            REM INITIALIZE INNER LOOP
130            WURDS$=""
140            SYMBOL$=LEFT$(LINE$,1)
150            REM REPEAT UNTIL SYMBOL$=" "
160            IF SYMBOL$=" " THEN GOTO 220
170                WURDS$=WURDS$+SYMBOL$
180                REM SHORTEN LINE BY SYMBOL
190                LINE$=MID$(LINE$,2)
200                SYMBOL$=LEFT$(LINE$,1)
210                GOTO 160
220            REM END OF LOOP
230            PRINT WURD$
240            IF LINE$<>" " THEN LINE$=MID$(LINE$,2)
250            GOTO 100
260        REM END OF LOOP
263        PRINT "TYPE LINE OF TEXT, MAX 40 CHAR"
265        INPUT LINE$
270        GOTO 040
280    REM END OF LOOP
290    END
```

This is not as efficient a program as it might be; by backing up the solution tree we might improve it. We will give another way to "split off and output next word".

```
REPEAT UNTIL LINE$=" "
    Find place of first blank in line
    Split off substring up to blank as word
    Remove word and trailing blank from line
```

This will result in a change in lines 090-290 to these

```
090     REM REPEAT UNTIL LINE$=" "
100     IF LINE$=" " THEN GOTO 230
110         PLACE=1
120         REM FIND PLACE OF FIRST BLANK IN LINE
130         REM REPEAT UNTIL MID$(LINE$,PLACE,1)=" "
140         IF MID$(LINE$,PLACE,1)=" " THEN GOTO 170
150             PLACE=PLACE+1
160             GOTO 140
170         REM END OF LOOP
180         REM SPLIT OFF SUBSTRING UP TO BLANK AS WORD
190         PRINT LEFT$(LINE$,PLACE-1)
200         REM REMOVE WORD AND TRAILING BLANKS
210         IF PLACE<LEN(LINE$)THEN LINE$=MID$(LINE$,PLACE+1)
220         GOTO 100
230     REM END OF LOOP
```

This algorithm is more efficient because in the inner REPEAT UNTIL loop (160 to 220 in first program; 130 to 170 in the second) there is only one call to a substring function in the second method whereas in the first there are three such calls as well as a concatenation. This inner loop is executed once for every character in the line so that its efficiency is important.

WORD STATISTICS

We have learned to read words of a text, recognize certain words such as STOP, replace words and to treat words in a list in different ways as we did with the list of names. Another important use of a computer in dealing with words involves keeping statistics about the lengths of words. Different authors have different patterns of use of words and this shows up in the frequency with which they use words of different lengths. Some authors use a lot of long words others rarely do.

In this section we will read a text and from it prepare a frequency distribution of word lengths. To add a little extra interest to this problem, we will display the results in graphic form. For instance, we will have a display like this for output:

```
LENGTH OF WORD          FREQUENCY
      1                 ***
      2                 ******
      3                 ********
      4                 *****
```

This display, called a histogram, represents the result of analyzing the frequency of different word lengths in a text. It shows that there were 3 one-letter words, 6 two-letter words, 8 three-letter words, and 5 four-letter words. The spacing in the display we show is not the same as what you will see on your screen because typesetting uses different widths for different characters. On your screen each character in the display occupies the same amount of space.

We will use what we have learned so far about reading words, but instead of forming the words we will just count the number of letters, then after each word concatenate an asterisk onto a string of asterisks stored in an array location corresponding to the letter count. We will use the array FRQUENCY$ to store the strings of asterisks; FRQUENCY$(4) will store a string of asterisks corresponding to the number of four-letter words. We will assume a maximum of twenty letters in a word and no punctuation marks. Here is the program:

```
010    REM PROGRAM DISPLAY
020    REM DETERMINE FREQUENCIES OF WORD LENGTHS
030    DIM FRQUENCY$(20)
040    REM INITIALIZE ARRAY TO NULL STRINGS
050    FOR I=1 TO 20
060        FRQUENCY$(I)=""
070        NEXT
075    PRINT "ENTER LINES, THEN ZZZ"
080    INPUT LINE$
090    REM REPEAT UNTIL LINE$="ZZZ"
100    IF LINE$="ZZZ" THEN GOTO 330
110        LINE$=LINE$+" "
120        SIZE=LEN(LINE$)
130        PLACE=1
140        REM REPEAT UNTIL PLACE >=SIZE
150        IF PLACE>=SIZE GOTO 300
160            REM REPEAT UNTIL MID$(LINE$,PLACE,1)<>" "
                   OR PLACE>=SIZE
170            IF MID$(LINE$,PLACE,1)<>" " OR PLACE>=SIZE
                   THEN GOTO 200
180                PLACE=PLACE+1
190                GOTO 170
200            REM END OF LOOP
210            START=PLACE
220            REM REPEAT UNTIL MID$(LINE$,PLACE,1)=" "
230            IF MID$(LINE$,PLACE,1)=" " GOTO 260
```

```
240             PLACE=PLACE+1
250             GOTO 230
260          REM END OF LOOP
270          COUNT=PLACE-START
280          IF COUNT<-20 AND COUNT>=1 THEN
                FRQUENCY$(COUNT)=FRQUENCY$(COUNT)+"*"
290          GOTO 150
300       REM END OF LOOP
310       INPUT LINE$
320       GOTO 100
330    REM END OF LOOP
340    PRINT "LENGTH OF WORD";TAB(20);"FREQUENCY"
350    FOR I=1 TO 20
360       PRINT TAB(10);I;TAB(20);FRQUENCY$(I)
370       NEXT
380    END
```

Here is a sample display:

```
ENTER LINES, THEN ZZZ
?ROSES ARE RED
?VIOLETS ARE BLUE
?HONEY IS SWEET
?AND SO ARE YOU
?ZZZ
      LENGTH OF WORD          FREQUENCY
                1
                2             **
                3             ******
                4             *
                5             ***
                6
                7             *
                ...
               20
```

READING BASIC

We have been reading English text and performing operations on it, or as the result of it. All these operations, we see, are absolutely mechanical but give you the impression that the computer is capable of doing things we think of as "intelligent work". One of the fields of artificial intelligence that has been explored is the translation from one language to another.

The translation of one natural language into another, such as English to French, has had only qualified success. It works, but not well when the text is ambiguous or difficult. The translations are not good literature, to say the least.

But computers are being used for language translation every day, for the translation of programming languages into machine languages. The reason this is possible is that programming languages are well defined and quite limited.

Your Basic programs are translated by the interpreter program one statement at a time into machine language before execution. A Basic program is made up of keywords and identifiers such as FOR, INPUT and IF and special symbols such as semicolon and colon. The Basic interpreter can tell which keyword it has read by comparing the word it has read to the known keywords. When a statement keyword such as GOTO is recognized, the translator knows it is to produce machine language for a jump.

CHAPTER 11 SUMMARY

In this chapter we have shown how the computer can, in a sense, understand English. The computer can recognize words by scanning for their beginnings and endings. Typically, words in English are surrounded by blanks or special characters; programs can be written which separate out words by searching for these characters. Due to the great speed of computers, they sometimes have the appearance of being intelligent, in spite of the fact that their basic mode of operation is very simplistic, such as seeing if a given character is a blank. Basic interpreters have been developed to read text which looks somewhat like English the text which they have been designed to read is Basic programs.

CHAPTER 11 EXERCISES

1. A *palindrome* is a word or phrase which is spelled the same backwards and forwards. The following are examples of word palindromes: "I", "mom", "deed" and "level". Blanks and punctuation are ignored in phrase palindromes, for example, "Madam, in Eden I'm Adam" and "A man, a plan, a canal, Panama". Write a program which reads a string and determines if it is a palindrome.

2. You are to write a program which will help in reviewing a script to determine its suitability for television screening. Your program is to give a list of the frequency of use of the following unacceptable words:

PHOOEY SHUCKS JEEPERS GOLLY

Make up a few lines of script to test out your program.

3. You are employed by an English teacher who insists that "and" should not be preceded by a comma. Hence, "Crosby, Stills, Nash and Young" is acceptable, but "Merril, Fynch, and Lynn" is not. Write a program which reads lines of text, searches for unacceptable commas, removes them, and makes stern remarks such as the English teacher would make about errors. (We are somewhat ambivalent about this!)

4. Write a program which reads text and then outputs it so that its left and right margins are vertical. First the program is to read the number of characters to output per line. Whenever enough words for one line have been collected, blanks are inserted between words to expand the line to the desired width. Then the expanded line is output.

Chapter 12

SUBPROGRAMS

In this chapter we will be introducing the idea of subprograms other than the main program that has constituted all programs so far. The purpose of having subsidiary programs or subprograms is so that a larger program can be divided into parts. In this way we "divide and conquer" a complicated problem. Sometimes a part of the solution of one problem can be used in many different problems, and making it into a subprogram creates a module or building block which can be used in other programs.

SUBROUTINES

There are two kinds of subprograms in Basic: subroutines and functions. Essentially, subroutines allow you to invent new Basic statements, while functions allow you to invent new operations. We will give an example of a subroutine first. This subroutine will find the index MAX of the largest element in an array ARRAY of N numbers. We will give the subroutine a name, MAXIMUM, just as we gave our programs names. The line numbers for the statements of a subroutine will, by convention, be chosen to be in the thousands. We begin this subroutine at line 5010. Here is the subroutine.

```
5010    REM SUBROUTINE MAXIMUM(ARRAY,N,MAX)
5020    REM ARRAY(MAX)IS LARGEST OF N ELEMENTS
5030    REM LOCAL VARIABLE: INDEX
5040    MAX=1
5050    FOR INDEX=2 TO N
5060        IF ARRAY(INDEX)>ARRAY(MAX)THEN MAX=INDEX
5070        NEXT
5080    RETURN
```

We will adopt a convention that our subroutines will all begin with a remark of the form

REM SUBROUTINE name of subroutine(parameters of subroutine)

In this way you can clearly identify that it is a subroutine and have a name to call it by. We have called our first subroutine MAXIMUM. In parentheses after the name there is a list of what are known as *parameters* of the subroutine. When we enter a subroutine from a main program we must have arranged that certain of these parameters, ARRAY and N, have been given values by the main program. Then, after we return to the main program from the subroutine, we will find the output from the subroutine in the parameter MAX. Before we look at an example of a main program that uses this subroutine, notice a few more points about the subroutine.

After the first remark that identifies it as a subroutine and lists the parameters, there is a remark that explains the purpose of the subroutine. Then follows a remark that lists the identifiers of any variables used in the subroutine that are not parameters. It is important to know the names of these *local variables* so that you do not choose the same names for variables in the main program. For instance, if we used I instead of INDEX then there might be a problem if I is in the main program. This remark listing the local variables saves you reading the subroutine to find them for yourself; it is part of good documentation. At the end of the subroutine is the statement RETURN. This returns control to the main program at the point where you left to enter the subroutine.

Here is the way to use this subroutine:

```
010    REM PROGRAM WINNER
020    DIM ARRAY(20)
030    PRINT "ENTER POSITIVE NUMBERS, END WITH -1"
040    N=1
050    INPUT ARRAY(N)
060    REM REPEAT UNTIL ARRAY(N)= -1
070    IF ARRAY(N)= -1 THEN GOTO 110
080        N=N+1
090        INPUT ARRAY(N)
100        GOTO 070
110    REM END OF LOOP
120    N=N-1
130    GOSUB 5010
140    PRINT "MAXIMUM NUMBER=";ARRAY(MAX)
150    END

5010   REM SUBROUTINE MAXIMUM(ARRAY,N,MAX)
           ...(as before)
5090   RETURN
```

The subroutine is *called* or *invoked* by using the statement GOSUB in line 130 of the main program, followed by the line number of the first statement of the subroutine. When the subroutine is entered, values have been given to the input parameters ARRAY and N. Return from the subroutine goes to the statement right after the GOSUB. At this stage a value has been given to the output parameter MAX and ARRAY(MAX) can be printed.

We have said that making a subroutine divides a larger program into smaller parts which are often called *modules.* This method of writing a larger program in smaller parts is called *modular programming.*

Another, perhaps more important, reason for using subroutines is that the same subroutine can be used again in another program. We will now show an example of how our MAXIMUM subroutine can be used to sort an array of positive numbers in ascending order. In this program we will again need to read in the array of positive numbers so we will write another subroutine for doing this. Here it is:

```
1010    REM SUBROUTINE ARRAYREAD(ARRAY,N)
1020    REM READ POSITIVE NUMBERS INTO ARRAY
1030    REM LOCAL VARIABLES:
1040    PRINT "ENTER POSITIVE NUMBERS, END WITH -1"
1050    N=1
1060    INPUT ARRAY(N)
1070    REM REPEAT UNTIL ARRAY(N)= -1
1080    IF ARRAY(N)= -1 THEN GOTO 1120
1090        N=N+1
1100        INPUT ARRAY(N)
1110        GOTO 1080
1120    REM END OF LOOP
1130    N=N-1
1140    RETURN
```

Here is the main program that uses the two subroutines and outputs the array of numbers in ascending sequence.

```
010     REM PROGRAM SORT
020     DIM ARRAY(20)
030     REM READ IN ARRAY
040     GOSUB 1010
050     LAST=N
060     FOR N=LAST TO 2 STEP -1
070         REM FIND MAXIMUM OF ARRAY(1) TO ARRAY (N)
080         GOSUB 5010
090         TEMP=ARRAY(N)
```

```
100         ARRAY(N)=ARRAY(MAX)
110         ARRAY(MAX)=TEMP
120         NEXT
130     REM PRINT SORTED ARRAY
140     FOR I=1 TO LAST
150         PRINT ARRAY(I)
160         NEXT
170     END

(copy subroutines ARRAYREAD and MAXIMUM here)
```

This is not a particularly efficient way to sort a list of numbers but it does illustrate the versatility of subroutines.

The GOSUB can appear in a case-like construct if you use the Basic statement

ON case number GOSUB lineA, lineB, ..., lineN

where the subroutine starting at line A is chosen if the case number has a value 1, the subroutine starting at line B if the case number has value 2, and so on.

PROBLEMS WITH SUBROUTINES

When subroutines are created for use in one particular main program there is no difficulty. The names of parameters can be chosen to fit in with the names of variables in the main program. Also line numbers of the statements in the various subroutines can be chosen so they do not conflict with each other. However, if a subroutine written for one main program is used in another it may be rather constraining to have to use the same set of variable names. Also you may have two different subroutines that you want to use and the names are inconsistent. This might have been the case for the two subroutines we have shown so far. For this reason, in most high-level languages it is possible to use a different set of names for the parameters of a subroutine when the subroutine is called by a program. These other names are called *arguments* and are listed in the same sequence as the parameter names they correspond to in the statement that calls the subroutine. One of the great weaknesses of Basic as a high-level language is the it does not have the argument-parameter mechanism for its subroutines. It does have it for its functions, however, as you will see.

Another problem about subroutines is that, although we start numbering their lines at some high number like 5010, so that they do not

conflict with the main program's line numbers they can conflict with each other. What you have to do is change the numbers yourself. If there are any GOTOs in the subroutine, the line numbers they specify must also be changed consistently.

Again, in most high-level languages relocation of subroutines is handled by the operating system.

FUNCTIONS

The function in Basic is not really a subprogram since it consists of only one statement. That statement is an assignment statement that assigns to the name of a function the value of an expression. Functions must be defined in the program before they are used. To define a function we use a Basic statement of the form

DEF FN name of function(name of parameter)=arithmetic expression

Functions are restricted to numerical values; the right-hand side of the assignment must be an arithmetic expression involving one variable name. This variable is the parameter of the function. For example:

DEF FN CUBE(A)=A*A*A

defines a function called CUBE whose value is the cube of the parameter A. If we were to include in the program the statement

PRINT FN CUBE(10)

then the value 1000 would be output. We invoke a function, by using the keyword FN, followed by the name of the function followed, in parentheses, by the argument of the function. Here the argument is a constant, the integer 10. When the statement is executed the argument is substituted for the parameter and the expression evaluated and the value assigned to CUBE.

Sometimes a parameter is called a *dummy argument* since it will be substituted by an argument whenever the function is invoked.

As another example of a function, here is a program that converts temperatures in Fahrenheit to temperatures in the Celsius scale.

```
010     REM PROGRAM METRIC
020     REM READ FARHENHEIT AND OUTPUT CELSIUS
030     DEF FN CELSIUS(FAHR)=(FAHR-32)*5/9
040     INPUT "TEMPERATURE IN FAHRENHEIT=";TEMP
050     PRINT "TEMPERATURE IN CELSIUS=";FN CELSIUS(TEMP)
060     END
```

Here is a sample display:

TEMPERATURE IN FAHRENHEIT=86
TEMPERATURE IN CELSIUS=30

The function is defined in line 030 before it is invoked in line 050. The argument of the invocation is TEMP and it is substituted for the parameter (or dummy argument) FAHR of the function definition.

Because functions in Basic are restricted to a single assignment statement they are not terribly useful. There are however a number of functions that are built into the Basic system.

BUILT-IN FUNCTIONS

We have already met several built-in functions in our discussion of handling alphabetic information. These were LEN the function whose value is the length of the string which is the value of its argument. For example, LEN("OUT"+"SIDE") has the value 7; the result of concatenating the two strings "OUT" and "SIDE" is a string of length 7. The functions we used to select substrings from strings were MID$, LEFT$, and RIGHT$. The arguments of these three functions have string values. The value of the function is also a string; that is why their names end with $. The value of LEN is a number; it does not end in $.

The difference between a user-defined function and a built-in function is that we do not need to precede the built-in's name with the keyword FN.

Other built-in functions that are useful in ordinary circumstances are:

ABS which returns the absolute value (that is, the value without sign) of its argument.

SGN which returns -1 if its argument is negative, 0 if the argument is zero, and 1 if the argument is positive.

SQR which returns the square root of its argument.

INT which returns the largest integer less than or equal to the value of its argument. For positive numbers this is equivalent to truncation. (We have often used INT.)

There are a number of mathematical functions used in scientific and engineering calculations. SIN gives the trigometric sine of its argument which must be an angle in radians. COS gives the cosine; TAN the tangent; ATN gives the angle in radians that has its argument as a tangent. EXP raises *e* to the power of the argument. LOG gives the natural logarithm of the value of the argument.

RANDOM NUMBERS

One of the most interesting built-in functions is RND. It produces a pseudo-random number from a series of numbers that are uniformly distributed between 0 and 1. If the argument is positive, a new random number in the series is generated each time the function is called. To start a series of pseudo-random numbers RND is invoked with a negative argument to begin. Each different starting negative argument provides a different *seed* for a series of pseudo-random numbers.

Here is an example where RND is used to give a series of dice rolls. Each of the two dice has a value between 1 and 6 with a uniform probability.

```
010    REM PROGRAM DICE
020    REM SIMULATES THE ROLL OF TWO DICE
030    INPUT "NUMBER OF ROLLS=";N
040    INPUT "START=";START
050    BEGIN=RND(-START)
060    FOR I=1 TO N
070        INPUT "PRESS Y TO ROLL";YES$
080        PRINT "ROLL IS ";INT(6*RND(1)+1)+INT(6*RND(1)+1)
090        NEXT
100    END
```

Positive numbers are entered for N and for START then N numbers of rolls are output. Before you get the value for the roll you must press Y in response to the prompt.

CHAPTER 12 SUMMARY

In this chapter we have introduced subprograms. Subprograms allow us to build up programs out of modules. The reasons for using subprograms in programs include the following:

1. Dividing the program into parts which can be written by different people.
2. Dividing a program into parts which can be written over a period of time.
3. Making a large program easier to understand by building it up out of conceptually simple parts.
4. Factoring out common parts of a program so they need not be written many times within a program.

5. Factoring out commonly-used logic so that it can be used in a number of different programs.
6. Separating parts of a program so they can be individually tested.

There are two kinds of subprograms in Basic: subroutines and functions. Essentially, a subroutine provides a new kind of Basic statement and a function provides a new kind of operation. The following important terms were discussed in this chapter.

Subroutine definition - the subprogram that defines the subroutine. The lines of a subroutine are given numbers higher than those of the main program. As a convention we begin each subroutine by three remarks which explain the subroutine. The first has the form

REM SUBROUTINE name of subroutine(list of parameters)

The second remark describes the purpose of the subroutine and the third lists the names of variables that are not parameters but are used locally inside the subroutine. It has the form

REM LOCAL VARIABLES: list of local variables

The last statement of a subroutine is RETURN.

Parameters of subroutine - variables that are used by the subroutine and that have been given values by the main program, the input parameters; or variables whose values are assigned by the subroutine and which will be used by the main program, the output parameters. In Basic the names of parameters of a subroutine must be the same as those used by the main program.

Calling (or invoking) a subroutine - placing in the main program the statement

GOSUB line number of first statement of subroutine

Control is passed to the subroutine by this statement and is returned to the main program at the statement following the GOSUB by the RETURN statement of the subroutine.

Local variables - the variables used in a subroutine which are not parameters. The names of these variables should not be the same as any referred to by the main program. Local variables are listed in a remark so that you can check that there is no possible confusion.

ON...GOSUB - a Basic statement that can be used in a case-like construct. Its form is

ON case number GOSUB lineA, lineB, ... lineN

where subroutines begin at line A, line B, etc. only one of which is executed depending on the value of case number.

Function definition - a function is defined by a statement of the form

DEF FN name of function(name of parameter) = arithmetic expression

The variable used in the arithmetic expression involving one variable is the parameter of the function. Functions consist of a single assignment statement and have one parameter whereas subroutines can have many statements and many parameters.

Invoking a function - a function can be used in any arithmetic statement using the form

FN name of function(argument)

The argument is any constant, numerical variable, or arithmetic expression that has a value. This value is substituted for the parameter in assigning a value to the function. For example,

```
DEF FN AREA(R) = (22/7)*R^2
A=5
PRINT AREA(2*A)
```

will output the area of the circle of radius 10 (that is, 2 times A which is 5)

User-defined function - a function defined by the programmer. Function definitions must precede statements in which they are used.

Built-in function - in Basic a number of functions have been predefined and are available. Some of these are for strings. User-defined functions are restricted to numerical values.

CHAPTER 12 EXERCISES

1. What does the following program output?

```
010    REM PROGRAM WEATHER
020    DIM TEMPERTURE(31)
030    INPUT "NUMBER OF DAYS=";TIME
040    PRINT "ENTER WEATHER FOR ";TIME;" DAYS"
050    PRINT "TEMPERATURE"
060    FOR DAY=1 TO TIME
070        INPUT TEMPERTURE(DAY)
080        NEXT
090    PRINT " AVERAGE TEMPERATURE:";
100    GOSUB 1010
110    END

1010   REM SUBROUTINE AVERAGE(TEMPERTURE,TIME)
1020   REM COMPUTES AVERAGE TEMPERATURE FOR TIME DAYS
1030   REM LOCAL VARIABLE: I
1040   SUM=0
1050   FOR I=1 TO TIME
1060       SUM=SUM+TEMPERTURE(I)
1070       NEXT
1080   PRINT SUM/TIME
1090   RETURN
```

2. Write a subroutine that reads an array of names counting how many it gets and another which sorts the array of names into alphabetic order. Now write the main program that uses the two subroutines.

3. What does the following short subroutine do? Write a small program which uses this short subroutine.

```
REM SUBROUTINE METRIC(SIZE)
SIZE=2.54*SIZE
RETURN
```

4. What does the following function do? Write a small program which uses this function.

```
DEF FN CONVERT(INCHES)=2.54*INCHES
```

Chapter 13

GRAPHICS

One of the most important extensions that Applesoft II Basic has over standard Basic is the ability to display graphic images. When we use PRINT statements in our Basic programs we are displaying characters on the screen. These characters are arranged in lines. There is a maximum of 40 characters in a line on the screen and a maximum of 24 lines. If you write a program that outputs more lines than can be held on the screen at any time, the image will scroll. This means that lines disappear from the top and new lines appear at the bottom. The screen is a *window* that lets us see only so much output or program listing at one time. In this chapter we will show how to write programs that draw lines that can be seen through the window of the screen. We call the line drawings graphics.

TEXT AND GRAPH MODES

There are three kinds of display available in Applesoft Basic; only one kind can be used at any one time. We can have lines of characters called *text mode,* or one of two kinds of graph mode: *low-resolution graph mode* or *high-resolution graph mode.*

So far we have been using the text mode. In text mode it is possible to create some effects that are different from the normal ones we are used to. We can, for instance, change the size of the text window. The instruction

POKE 32,*x*

sets the left margin of the window to position *x*. The value of *x* can be 0 to 39.

POKE 33,*w*

sets the window size to *w* characters. The normal size is 40 characters.

POKE 34,*h*

sets the top of the window to line *h*. The range is 0 to 23.

POKE 35,*b*

sets the bottom of the window to line *b*. You must not set the bottom higher than the top.

Another instruction

CALL - 912

moves the text one line up on the screen. The *Basic Programming Reference Manual* has more such instructions detailed.

Another thing you can do is to produce a black on white display rather than white on black. To do this give the command

INVERSE

To cause the display to flash use the command

FLASH

To return to a normal mode use

NORMAL

That is enough for text mode; we will now move to graphics.

To begin we will discuss low-resolution graphics. To enter the low-resolution graph mode we use the Basic statement GR. If you give this as an immediate execution instruction you will see that the text disappears except for 4 lines at the bottom of the screen; a graphics window is established that is 40 units wide and 40 units high. The window will be dark.

CHOOSING A COLOR FOR LOW-RESOLUTION GRAPHICS

The color used in drawing on the screen is set by a statement of the form

COLOR=arithmetic expression

The value of the arithmetic expression should be an integer between 1 and 15 inclusive. The meaning of these is:

0 black	4 dark green	8 brown	12 green
1 magenta	5 grey	9 orange	13 yellow
2 dark blue	6 medium blue	10 grey	14 aqua
3 purple	7 light blue	11 pink	15 white

If you do not have a color display you will be limited to 0 and 15: black and white. You can experiment with color if you have a color monitor.

PLOTTING A POINT ON THE SCREEN

When we plot a point on the screen using low-resolution graphics we actually get a rather sizable brick-shaped rectangle of whatever color we have set for plotting by the COLOR statement. The position of this brick of color is located in the graphics window by two numbers, called *coordinates.* The first coordinate (called the *x*-coordinate) is the number of units along the screen from the top left-hand corner to a place vertically above where the brick will be. The second coordinate (called the *y*-coordinate) is the distance vertically down to the brick. Coordinates must be integers. We can plot a point (a brick) by the statement

PLOT *x*-coordinate, *y*-coordinate

To plot a brick in the upper left-hand corner of the graphics window we use

PLOT 0,0

To plot one in the upper right-hand corner we use

PLOT 39,0

The values of the *x*-coordinates range from 0 to 39; there are 40 possible *x*-values. A brick in the lower right-hand corner is formed by

PLOT 39,39

Whenever we finish using graphs we can return to the text mode by giving the Basic instruction TEXT. If you do this at the end of your program, your graphic design will disappear. You can always give it as an immediate execution statement when you have finished with the program that produces the graphic. Sometimes it is also neater to move the cursor to the top of the screen by giving the immediate execution instruction HOME.

Here is a program that fills the screen with the color of your choice

```
010    REM PROGRAM COLORSCREEN
020    REM FILLS WINDOW WITH COLOR
030    TEXT
040    INPUT "ENTER COLOR(1-15):";BACKROUND
050    GR
060    COLOR=BACKROUND
070    REM COLOR SCREEN BRICK BY BRICK
080    FOR X=0 TO 39
090        FOR Y=0 TO 39
100            PLOT X,Y
110            NEXT
120        NEXT
130    END
```

Here is a program to do vertical stripes:

```
010    REM PROGRAM STRIPES
020    REM DRAWS VERTICAL STRIPES
030    TEXT
040    INPUT "ENTER BACKGROUND COLOR(1-15):";BACKROUND
050    INPUT "ENTER FOREGROUND COLOR(1-15):";FOURROUND
060    GR
070    COLOR=BACKROUND
080    FOR X=0 TO 38 STEP 2
090        FOR Y=0 TO 39
100            PLOT X,Y
110            NEXT
120        NEXT
130    COLOR=FOURROUND
140    FOR X=1 TO 39 STEP 2
150        FOR Y=0 TO 39
160            PLOT X,Y
170            NEXT
180        NEXT
190    END
```

Notice that we could not use the identifiers BACKGROUND or FOREGROUND because of the keywords GR and OR. This same program can be done another way as you will see in the next section.

DRAWING LINES ON THE SCREEN

In the last example we had a series of vertical stripes. We can plot such stripes by an instruction of the form

VLIN *y*-value of line start, *y*-value of line finish AT *x*-value for line

A vertical line going from the point (0,0) to (0,39) is produced by

```
VLIN 0,39 AT 0
```

Here is the stripe program using VLIN. We will just do one set of stripes.

```
010    REM PROGRAM BARS
020    REM DRAWS WHITE VERTICAL BARS
030    GR
040    COLOR=15
050    FOR X=0 TO 38 STEP 2
060        VLIN 0,39 AT X
070        NEXT
080    END
```

A similar instruction draws horizontal lines. The instruction has the form

HLIN *x*-value at start, *x*-value at end AT y-value for line

Here is a subroutine for drawing a rectangle of length L and height H with its top left-hand corner at point (X,Y). We will choose the color and go into graph mode in the main program.

```
5010    REM SUBROUTINE RECTANGLE(L,H,X,Y)
5020    REM DRAWS RECTANGLE OF LENGTH L AND HEIGHT H
5030    REM UPPER LEFT CORNER AT (X,Y)
5040    HLIN X,X+L AT Y
5050    HLIN X,X+L AT Y+H
5060    VLIN Y,Y+H AT X
5070    VLIN Y,Y+H AT X+L
5080    RETURN
```

Here is a main program that uses this subroutine

```
010     REM PROGRAM DRAWRECT
020     REM DRAWS A SERIES OF RECTANGLES
030     ONERR GOTO 150
040     GR
045     COLOR=15
047     REM INFINITE LOOP
050         INPUT "L=";L
060         INPUT "H=";H
070         INPUT "X=";X
080         INPUT "Y=";Y
095         REM CALL SUBROUTINE RECTANGLE
100         GOSUB 5010
110         GOTO 050
120     END

150     REM DEAL WITH ERROR
160     ERR=PEEK(222)
170     IF ERR=53 GOTO 050
180     PRINT "UNEXPECTED ERROR:";ERR
190     STOP
```

If we try to plot points or lines outside the graphics window an error results which would give the message

?ILLEGAL QUANTITY ERROR

This error has code 53. We do not want to stop if this happens but want to choose a new set of values for another rectangle. That is why we have included the ONERR GOTO instruction. Try modifying this program so that each rectangle can be in the color of your choice.

GAME CONTROLS AND GRAPHICS

Attached to the Apple computer are two game controls called *paddles.* With each paddle there are two things you can do: either press the button or twist the knob. The two paddles are numbered, usually 0 and 1. The button of a paddle is either pressed or not pressed and its state can be discovered in the program by means of a PEEK. If button #0 is being pushed, then the value of

PEEK (-16287)

is greater than 127. Similarly,

PEEK (-16286)

has a value greater than 127 if button #1 is being pushed.

As far as graphics are concerned the function

PDL (number of paddle)

is of more interest. This function has a value from 0 to 255 depending on the position of the knob on the paddle. We can use the paddles to control the x and y positions of bricks plotted on the screen.

Here is a program that lets you draw on the screen using the paddle controls.

```
010    REM PROGRAM SKETCH
020    REM ACTS LIKE A SKETCH PAD
030    GR
040    COLOR=15
050    SKALE=39/255
055    REM INFINITE LOOP
060        X=PDL(0)*SKALE
070        Y=PDL(1)*SKALE
080        PLOT X,Y
090        GOTO 060
100    END
```

We had to scale the paddle readings to produce a number for X and Y between 0 and 39 instead of between 0 and 255 which is the actual value of the paddle reading.

MAKING SOUNDS ON THE COMPUTER

This chapter is about graphics but sometimes it is rather interesting to have sound accompanying a picture particularly if it is a moving picture. We will discuss animation in the next section and in the example will have instructions that cause the computer to make a sound. In this book we will not discuss making music; that is a whole subject in itself. Our sounds will just be clicks or beeps on the loudspeaker attached to the computer. We can communicate with the speaker by PEEKing at the machine address that is related to the speaker. The function

PEEK (-16336)

must be invoked to stimulate the speaker to make a click. (It is rather difficult to believe that a negative number is related to a machine location but it is.) To achieve some sort of noticeable sound you must invoke the function a number of times in quick succession. Try this program.

```
010    REM PROGRAM SPEAKER
020    REM STIMULATE SPEAKER
030    S=-16336: REM SPEAKER ADDRESS
035    REM INFINITE LOOP
040        INPUT "CHOOSE A SOUND (TYPE ANY INTEGER)";N
050        REM MAKE THAT SOUND
060        FOR I=1 TO N
070            SOUND=PEEK(S)
080            NEXT
090        GOTO 040
100    END
```

Notice that we assigned the peculiar value -16336 to the variable S. We can experiment with this program by assigning to SOUND a more complicated combination such as

SOUND=PEEK(S)-PEEK(S)+PEEK(S)-PEEK(S) and so on

Try changing line 070 amd run the program again. You could even make a new electronic musical instrument by having different subroutines for making sounds called when you type different integers. You can use this kind of input sequence for your program

```
010    REM PROGRAM SYNTHESIZER
020    S=-16336: REM SPEAKER ADDRESS
025    REM INFINITE LOOP
030        INPUT "TYPE A NOTE (1-12)"; NOTE
040        ON NOTE GOSUB label1, label2, ... label12
050        GOTO 030
060    END
```

(12 subroutines making 12 sounds)

Have fun!

ANIMATION WITH GRAPHICS

The illusion of continuous motion can be created by having a series of still pictures in which an object is shown in slightly different positions from one to the next. We can write a program in which one of our bricks is moved around the screen. We will start the brick at the center of the screen and move it by using the RND function for generating a random change from its present position. When the brick reaches the edge of the window we will start it at the center again. Here is the program. The brick will move randomly like smoke particles in the air which is called Brownian movement.

```
010    REM PROGRAM BROWNIAN
020    REM PARTICLE STARTS AT CENTER AND MOVES RANDOMLY
030    INPUT "BEGIN=";BEGIN
035    REM BEGIN SERIES OF RANDOM NUMBERS
040    X=RND(-BEGIN)
045    ONERR GOTO 210
049    GR
050    COLOR=15
051    REM DRAW BORDERS
052    VLIN 0,39 AT 0
053    VLIN 0,39 AT 39
054    HLIN 0,39 AT 0
055    HLIN 0,39 AT 39
056    REM INFINITE LOOP
060        REM START AT CENTER
070        X=19
080        Y=19
090        REM START ANIMATION LOOP
095        REM LOOP UNTIL PARTICLE IS OUTSIDE WINDOW
100            COLOR=15
110            PLOT X,Y
115            REM COMPUTE AND PLOT NEXT POSITION
120            NX=X+INT(1.5-RND(1)*2)
130            NY=Y+INT(1.5-RND(1)*2)
140            PLOT NX,NY
145            REM ERASE PREVIOUS POSITION
150            COLOR=0
```

```
160          PLOT X,Y
165          REM MAKE NEXT POSITION INTO PREVIOUS POSITION
170          X=NX
180          Y=NY
190          GOTO 100
200     END

210     REM ERROR RESTART
220     ERR=PEEK(222)
221     REM MAKE A SOUND
222     S=-16336: REM SPEAKER ADDRESS
223     FOR I=1 TO 20
224         BANG=PEEK(S)-PEEK(S)+PEEK(S)-PEEK(S)
225         NEXT
230     IF ERR=53 THEN GOTO 060: REM RESTART AT CENTER
240     PRINT "UNEXPECTED ERROR=";ERR
250     STOP
```

You will notice that when the particle hits the lower border of the window it produces a path of characters and no sound unless it goes right to the bottom of the screen. The four text lines at the bottom of the screen are really part of the graphics window. We can convert to a full-screen graphics (40x48 positions) by placing the instruction

```
POKE -16302,0
```

after the GR instruction. You will not have any text lines then.

HIGH-RESOLUTION GRAPHICS

The graphics produced so far are rather coarse; the lines were fat. We can use a finer line with high-resolution graphics. To use these give the command

```
HGR
```

This produces a graphics window which is 280 by 160 units with four lines for text. The origin of coordinates is still in the upper left corner of the screen but now the *x*-coordinate can range vrom 0 to 279, the *y*-coordinate from 0 to 159. You can have a full screen window (280 by 192 units) by giving the instruction

```
POKE -16302, 0
```

the same as with the low-resolution graphics. The colors for high-resolution graphics are more limited; they are set by

HCOLOR=value

where value can be 0 to 7 inclusive. The colors are:

0 black1	4 black2
1 green	5 variable
2 blue	6 variable
3 white1	7 white2

If you have a non-color monitor choose either 3 or 7 for white. The actual color you get with a color TV monitor is variable. Try yours to find out what happens.

PLOTTING LINES FROM POINT TO POINT

The instruction HPLOT can be used to: plot a point, to plot a line from the last plotted point to the next, or to plot a line between two points. The form is

HPLOT x_1, y_1 TO x_2, y_2

This plots a line from (x_1,y_1) to (x_2,y_2). If TO x_2,y_2 is missing a point is plotted at (x_1,y_1). If (x_1,y_1) is missing a line is drawn from the last point plotted to (x_2,y_2).

This means that lines need not be restricted to vertical and horizontal lines as they tend to be in low-resolution graphics. Here is a rectangle-drawing subroutine for high-resolution graphics. In this subroutine we have an angle ALPHA specified in degrees. ALPHA is the angle between the top side of the rectangle and the horizontal. The top left corner of the rectangle is at (X, Y) and the length of the top side is L the other side is H. (The top side is the side starting at the upper left corner, which is the corner highest on the screen, and going to the right from there.)

This program uses trigonometry. If you do not understand it, never mind; you can use it anyway.

```
4010  REM SUBROUTINE TILTRECT(L,H,X,Y,ALPHA)
4020  REM PLOTS A TILTED RECTANGLE
4030  REM LOCAL VARIABLES: PI,RALPHA,C,S
4040  PI=3.14159
4050  RALPHA=ALPHA*PI/180
4060  C=COS(RALPHA)
4070  S=SIN(RALPHA)
4080  HPLOT X,Y
4090  HPLOT TO X+L*C,Y+L*S
4100  HPLOT TO X+L*C-H*S,Y+L*S+H*C
4110  HPLOT TO X-H*S,Y+H*C
```

```
4120    HPLOT TO X,Y
4130    RETURN
```

The COS and SIN functions require the angle to be in radians.

Here is a main program that lets you use this subroutine. Try it.

```
010     REM PROGRAM DRAWHIGHRES
020     REM DRAWS RECTANGLES TO YOUR SPECIFICATIONS
030     ONERR GOTO 150
040     HGR
045     HCOLOR=3
048     REM INFINITE LOOP
050         INPUT "L=";L
060         INPUT "H=";H
070         INPUT "X=";X
080         INPUT "Y=";Y
090         INPUT "ALPHA=";ALPHA
110         REM CALL SUBROUTINE TILTRECT
120         GOSUB 4010
130         GOTO 050
140     END

150     REM DEAL WITH ERROR
160     ERR=PEEK(222)
170     IF ERR=53 GOTO 050
180     PRINT "UNEXPECTED ERROR:";ERR
190     STOP
```

Remember that the scale of X and Y is different for high-resolution graphics so that the specifications for the rectangle's size will be quite different from what is was with the program DRAWRECT.

PLOTTING GRAPHS

Because the origin of coordinates is at the upper left of the screen instead of the lower left, the *x*-, *y*- coordinates normally used in geometry to produce graphs would produce upside-down graphs. To correct this, use high-resolution graph coordinates X, Y that are related to geometry coordinates *x, y* by

$$X = x$$

and

$$Y = 159 - y$$

Here is a program that plots a graph for the curve

$$y=x^2-x-2$$

between the values of $x=-2$ and $x=3$, at intervals of 0.2. There are 26 points in all. We will use a subroutine GRAPHPLOT which accepts N points X(I),Y(I) and plots a graph. This subroutine in turn uses two other similar subroutines XMINMAX and YMINMAX to find the minimum and maximum values of X and Y.

```
3010   REM SUBROUTINE XMINMAX(X,N,XLITTLE,XBIG)
3020   REM DETERMINES MINIMUM AND MAXIMUM X
3030   REM LOCAL VARIABLES: K
3040   XBIG=X(1)
3050   XLITTLE=X(1)
3060   FOR K=2 TO N
3070      IF X(K)>XBIG THEN XBIG=X(K)
3080      IF X(K)<XLITTLE THEN XLITTLE=X(K)
3090      NEXT
3100   RETURN

4010   REM SUBROUTINE YMINMAX(Y,N,YLITTLE,YBIG)
4020   REM DETERMINES MINIMUM AND MAXIMUM Y
4030   REM LOCAL VARIABLES: K
4040   YBIG=Y(1)
4050   YLITTLE=Y(1)
4060   FOR K=2 TO N
4070      IF Y(K)>YBIG THEN YBIG=Y(K)
4080      IF Y(K)<YLITTLE THEN YLITTLE=Y(K)
4090      NEXT
4100   RETURN
```

In a programming language where we have parameter-argument correspondence for subroutines the same subroutine would work for X or Y. Unfortunately, although Basic has parameter-argument correspondence for functions, it does not have it for subroutines.

Here is the GRAPHPLOT subroutine

```
2010   REM SUBROUTINE GRAPHPLOT(X,Y,N)
2020   REM PLOTS A GRAPH OF X VERSUS Y
2030   REM LOCAL VARIABLES: XLITTLE,XBIG,YLITTLE,YBIG
2040   REM XRANGE,YRANGE,YZERO,XZERO,YPLACE,XSKALE,YSKALE
2050   HGR
2060   HCOLOR=3
2070   REM CALL SUBROUTINE XMINMAX
```

```
2080    GOSUB 3010
2090    REM CALL SUBROUTINE YMINMAX
2100    GOSUB 4010
2110    YRANGE=YBIG-YLITTLE
2120    YSKALE=159/YRANGE
2130    YZERO=(0-YLITTLE)*YSKALE
2140    REM PLOT X-AXIS IF ON SCREEN
2150    YPLACE=159-YZERO
2160    IF YPLACE<159 AND YPLACE>0 THEN
                HPLOT YPLACE, 0 TO YPLACE,159
2170    XRANGE=XBIG-XLITTLE
2180    XSKALE=279/XRANGE
2190    XZERO=(0-XLITTLE)*XSKALE
2200    REM PLOT Y-AXIS IF ON SCREEN
2210    IF XZERO<280 AND XZERO>0 THEN
                HPLOT 0,XZERO TO 280, XZERO
2220    REM PLOT POINTS
2230    HPLOT (X(1)-XLITTLE)*XSKALE,
                159-((Y(1)-YLITTLE)*YSKALE)
2240    FOR J=2 TO N
2250        HPLOT TO (X(J)-XLITTLE)*XSKALE,
                159-((Y(J)-YLITTLE)*YSKALE)
2260        NEXT
2270    RETURN
```

Note the origin change is the cause of the different handling for the *y*-coordinates from the *x*-coordinates.

Here is the main program.

```
010     REM PROGRAM PARABOLA
020     REM PLOTS GRAPH OF A CURVE
030     DIM X(50), Y(50)
040     SPACING=.2
045     N=26
050     FOR I=1 TO N
060         X(I)=-2+(I-1)*SPACING
070         Y(I)=X(I)^2-X(I)-2
080         NEXT
090     REM CALL GRAPHPLOT SUBROUTINE
100     GOSUB 2010
110     END
```

The problem with graph plotting using this method is that we have no way of labelling the graph. You can, however, obtain software for the

Apple computer which will produce labelled graphics.

Another facility exists in high-resolution graphics involving *shapes* but this is beyond the scope of this book. Read the manual.

CHAPTER 13 SUMMARY

In this chapter we introduced the Applesoft extension to standard Basic for graphics. The normal display mode, called text mode, allows you to display 24 lines of characters, each line being up to 40 characters long. The computer can be switched to one of two graphics modes: low-resolution graphics and high-resolution graphics. In the graphics mode, 4 lines of text can still be displayed at the bottom of the screen. The following terms were discussed.

Window - the area of the screen in which either text or graphics is displayed. The size of the window is measured in different units for the three modes. In the text mode the window is 40 characters wide and 24 lines high. In low-resolution graphics the window is normally 40 units wide and 40 units high. It can be enlarged to 40 by 48 units by eliminating the text lines at the bottom. In high-resolution graphics the normal window is 280 units wide and 160 units high. It can be enlarged to 192 units high.

Scroll - the movement of lines of characters on the screen in text mode as additional lines are added to an already full screen. The top line disappears as all lines are moved up one to make room for a new line at the bottom.

Text mode - the mode in which lines of characters can be displayed. Graphs can be plotted in this mode as can be seen in Chapter 18.

Low-resolution graphics - a mode for producing coarse graphics. A plotted point appears as a brick-shaped rectangle on the screen. Horizontal and vertical lines (bars) can easily be plotted.

GR - the Basic statement that causes you to enter the low-resolution graphics mode. The graphics window is cleared by this statement.

TEXT - the statement (or command) that causes the machine to return to normal text mode.

COLOR - a variable that can be assigned an integer value between 0 and 15

inclusive. This determines the color of points or lines plotted in low-resolution graphics.

Origin of coordinates - the upper left-hand corner of the graphics window. The first (or *x*) coordinate is the distance in window units a point is from the left side of the screen. The second (or *y*) coordinate is the distance a point is *down* from the top of the screen. In geometry the origin is usually at the bottom left.

PLOT *x*, *y* - plot a point (brick shape) at the place on the screen with graphics coordinates (*x*,*y*).

HLIN...AT - plots a horizontal (broad) line at a particular *y*-level given after the AT between *x*-values given, separated by a comma, before the AT.

VLIN...AT - plots a vertical (broad) line at a particular *x*-level given after the AT between *y*-values given, separated by a comma, before the AT.

Game control - an input device attached to the computer which has a button that can be pressed and a knob that can be twisted. The state of the button (either pressed or not pressed) can be determined by the function PEEK(-16287) for game control #0 or PEEK(-16286) for game control #1. If the function has a value greater than 127 the button is pressed.

PDL(*n*) - a built-in function whose value is the knob setting on the game control (paddle) number *n*. The setting varies between 0 and 255.

Speaker - attached to the computer. When the function PEEK(-16336) is used in a program a click is produced on the speaker. Various sounds can be achieved by stimulating the speaker at regular intervals as in a loop.

Animation - showing a succession of still graphic images that can create the illusion of motion if they differ only slightly from each other.

High-resolution graphics - a mode for producing fine detail graphics. Lines can be drawn from one point to another.

HGR - the statement that puts the computer in the high-resolution graphics mode. The graphics window is cleared.

HCOLOR - the variable that controls the color in high-resolution graphics. Its value must be an integer between 0 and 7 inclusive.

HPLOT...TO - used to plot a point or a line from point to point.

Graphing a mathematical function - plotting lines joining corresponding values of x and y for a succession of values of x. Usually we plot values for which x increases uniformly.

Labelling of graphics - not possible with Applesoft graphics.

CHAPTER 13 EXERCISES

1. Use low-resolution graphics to plot a mathematical function. In this case draw a vertical line at a succession of equally-spaced values of x. The height of the line is to be the value of y for that x. Draw such a graph for $x=0$ to $x=30$ at intervals of 1 for the curve

   $$y=ax^2 + bx + c$$

   You are to read in values of a, b, and c, all positive. Then compute the 31 values of y, find the maximum value and scale all values so the maximum will have the full window height of 40 units. Now plot all the vertical bars. Use a subroutine to plot the bars.

2. Write a program using low-resolution graphics that places a succession of bricks on the screen in random positions in the 40x40 window and having random colors (code between 0 and 15). Pretty!

3. Write a subroutine in high-resolution graphics to draw a square of side s with its center at (x,y). Test it by having a main program draw a succession of squares of larger and larger size centered on the point (140,95). Have the program stop using an ONERR GOTO statement. Watch the bottom of the screen.

4. Test the high-resolution graphics function plotting subroutines given in this chapter by writing a main program to plot the curve

   $$y=\sin(x)$$

   between $x=0$ and 2Π radians, where $\Pi=3.14159$.

5. Write a program to sketch using the game controls and high-resolution graphics.

Chapter 14

SEARCHING AND SORTING

When a large amount of information is stored in a computer, it must be organized so that you are able to get at the information to make use of it. This problem of *data retrieval* is at the heart of all business operations. Records are kept of employees, customers, suppliers, inventory, in-process goods, and so on. These records are usually grouped in some way into what are called *files*. We might have, for example, a file of employee records, a file of customer records, an inventory file, and so on. Each file must be kept up to date.

A file that we all have access to is printed in the telephone book. It consists of a series of *records* of names, addresses, and telephone numbers. We say that there are three *fields* in each of these records: the name field, the address field, and the phone-number field. The file is in the alphabetic order of one of the three fields, the name field. We say that the name field is the *key* to the ordering of the file. The file is in alphabetic order on this field because that is how it can be most useful to us for data retrieval. We know someone's name and we want his phone number. We might also want his address and that too is available. The telephone company also has the same set of records, ordered using the phone-number field as the key.

In this chapter we will be investigating how a computer can search for information in a file and how records can be sorted.

LINEAR SEARCH

One way to look for data in a file is to start at the beginning and examine each record until you find the one you are looking for. This is the method people use who do not have large files. But for more than about 12 records it is not a good filing system. It will serve as an example to introduce us to the idea of searching mechanically and give us a bad method to compare our better methods to. We will create a file which consists of names and telephone numbers but the file will not be ordered by either name or number.

We will keep the file in two one-dimensional arrays, one called NAME$ and one called NUMBER$. The array for the phone numbers is alphabetic because the numbers consist of 3 digits, a dash, then 4 digits. NUMBER$(I) will be the correct telephone number for NAME$(I). We will read this file which is terminated by a dummy name, ZZZ, then read a list of names of people whose phone numbers are wanted. Here is the program to do this job. We are assuming that our file of names and phone numbers is entered so that the name is entered followed by a comma then the phone number. Because the names have commas in them we will have to enclose them in quotation marks.

```
010    REM PROGRAM PHONES
020    REM LOOK UP NUMBERS IN A DIRECTORY
030    DIM NAME$(50)
040    DIM NUMBER$(50)
050    REM INPUT FILE OF NAMES AND NUMBERS
060    PRINT "ENTER PHONE DIRECTORY"
070    PRINT "NAME";TAB(20);"NUMBER"
080    I=1
090    INPUT NAME$(I),NUMBER$(I)
100    REM REPEAT UNTIL NAME$(I)="ZZZ"
110    IF NAME$(I)="ZZZ" THEN GOTO 150
120        I=I+1
130        INPUT NAME$(I),NUMBER$(I)
140        GOTO 110
150    REM END OF LOOP
160    FILESIZE=I -1
170    INPUT "ENTER FRIEND'S NAME ";FRIEND$
180    REM REPEAT UNTIL FRIEND$="ZZZ"
190    IF FRIEND$="ZZZ" THEN GOTO 370
200        REM LOOK UP FRIEND'S NUMBER
210        I=1
220        REM REPEAT UNTIL NAME$(I)=FRIEND$ OR I>FILESIZE
230        IF NAME$(I)=FRIEND$ OR I>FILESIZE THEN GOTO 260
240            I=I+1
250            GOTO 230
260        REM END OF LOOP
270        REM IF FRIEND$=NAME$(I) THEN...ELSE
280        IF FRIEND$=NAME$(I) THEN GOTO 320
290            REM DO ELSE
300            PRINT FRIEND$;TAB(20);"UNLISTED"
310            GOTO 340
320            REM DO THEN
```

```
330           PRINT FRIEND$;TAB(20);NUMBER$(I)
340       REM END OF SELECTION
350       INPUT "ENTER FRIEND'S NAME ";FRIEND$
360       GOTO 190
370    REM END OF LOOP
380    END
```

Here is a sample display:

```
ENTER PHONE DIRECTORY
NAME                          PHONE
? "PERRAULT,R."          ,    "483-4865"
? "BORODIN,A."           ,    "782-8928"
? "COOK,S.A."            ,    "763-3900"
? "ENRIGHT,W.H."         ,    "266-1234"
? "ZZZ"                  ,    "999-9999"
ENTER FRIEND'S NAME "BORODIN,A."
BORODIN,A.                    782-8928
ENTER FRIEND'S NAME "DAVIES,R."
DAVIES,R.                     UNLISTED
ENTER FRIEND'S NAME "ZZZ"
```

TIME TAKEN FOR SEARCH

In the last section we developed a program for a linear search. The searching process consists of comparing the friend's name, FRIEND$, with each name in the file of names NAME$(1), NAME$(2), NAME$(3), and so on until either the name is found or the end of the file is reached. For a small file, a linear search like this one may be fast enough, but it can be time-consuming if the file is lengthy.

If there are N records in the file and the name is actually in the file, then on the average there will be N/2 comparisons. The largest number of comparisons would be N if the name were last in the file, the least number would be 1 if the name were first. A file of 1000 names would require 500 comparisons on the average. This gets to look rather formidable. It is for this reason that we do something to cut down on the effort. What we do is to sort the file into alphabetic order and then use a method of searching called *binary searching.* We will look at sorting later, but first we will see how much faster binary searching can be.

BINARY SEARCH

The telephone book is sorted alphabetically and the technique most of us use for looking up numbers is similar to the technique known as binary searching. We start by opening the book near where we think we will find the name we are looking for. We look at the page that is open and compare any name on it with the name being sought. If the listed name is alphabetically greater we know we must look only between the page we are at and the beginning of the book. We have eliminated the second part of the book from the search. This process is repeated in the part that might contain the name until we narrow the search down to one page.

In binary searching, instead of looking where we think we might find the name, we begin by looking at the name in the middle of the file and discard the half in which it cannot lie. This process cuts the possible number of names to be searched in half at each comparison.

A file of 16 names would require a maximum of 4 comparisons: one to cut the list to 8, another to 4, another to 2, and another to 1. Of course, we might find it earlier, but this is the *most* work we have to do. It is the maximum number of comparisons. With a linear search of 16 records we might have to make 16 comparisons, although 8 is the average. If we have a file of 1024 records, the binary search takes a maximum of 10 comparisons. This can be calculated by seeing how many times you must divide by 2 to get down to 1 record. Put mathematically, 1024 is equal to

```
2*2*2*2*2*2*2*2*2*2
```

Just one more comparison, making 11 altogether, will let you search a list of 2048 entries. Then 4096 can be done with 12 comparisons. You can see how much more efficient binary searching can be when the file is a long one.

A SUBROUTINE FOR BINARY SEARCH

We will now design a program for doing a binary search and write it so that it can be called as a subroutine. To use the subroutine, which we will call SEARCH, the file must be stored in an array called FILE$ whose length is SIZE. We will set the variable PLACE to the index in FILE$ where KEY$ is found. This means that FILE$(PLACE) will be equal to KEY$. If the KEY is not in the file, PLACE will be set to zero.

We will develop the algorithm for the binary search in two stages as an illustration of step-by-step refinement. We will write out our proposed solution in a form that is a mixture of English and Basic.

```
Set PLACE to zero in case KEY$ is not in FILE$
REPEAT UNTIL(there is no more of the file to search)
        Find middle of file
        IF middle value matches KEY$ THEN
                Set PLACE to middle
                Discard remainder of file
        ELSE
            IF middle value comes after KEY$ THEN
                Discard last half of remainder of file
            ELSE
                Discard first half of remainder of file
    END
```

It will be important to know the FIRST and LAST of the remainder of the file at any time in order to establish the MIDDLE and to discard the appropriate half. We initially set FIRST to 1 and LAST to SIZE. Then to find the middle we use

```
MIDDLE=INT((LAST+FIRST)/2)
```

It will not matter that this division is truncated as the process of finding the middle is approximate when the number of entries in the file is an even number. Refining the expression, "Discard last half of remainder of file," becomes

```
LAST=MIDDLE-1
```

and, "Discard first half of remainder of file," becomes

```
FIRST=MIDDLE+1
```

Notice that we are discarding FILE$(MIDDLE) as well in each case. The subroutine can now be written:

```
9010    REM SUBROUTINE SEARCH(FILE$,KEY$,SIZE,PLACE)
9020    REM FIND PLACE WHERE KEY$ IS IN FILE$
9030    REM USE BINARY SEARCH
9040    REM LOCAL VARIABLES:FIRST,LAST,MIDDLE
9050    REM SET PLACE TO ZERO IF KEY$ NOT IN FILE$
9060    PLACE=0
9070    REM INITIALIZE THE SEARCH LOOP
9080    FIRST=1
9090    LAST=SIZE
9100    REM REPEAT UNTIL FIRST>LAST
9110    IF FIRST>LAST THEN GOTO 9330
9120        MIDDLE=INT((FIRST+LAST)/2)
9130        REM IF FILE$(MIDDLE)=KEY$ THEN...ELSE
9140        IF FILE$(MIDDLE)=KEY$ THEN GOTO 9270
```

```
9150            REM DO ELSE
9160            REM IF FILE$(MIDDLE)>KEY$ THEN...ELSE
9170            IF FILE$(MIDDLE)>KEY$ THEN GOTO 9220
9180                REM DO ELSE
9190                REM DISCARD FIRST HALF
9200                FIRST=MIDDLE+1
9210                GOTO 9250
9220                REM DO THEN
9230                REM DISCARD LAST HALF
9240                LAST=MIDDLE -1
9250            REM END OF SELECTION
9260            GOTO 9310
9270            REM DO THEN
9280            PLACE=MIDDLE
9290            REM DISCARD ALL OF FILE
9300            FIRST=LAST+1
9310        REM END OF SELECTION
9320        GOTO 9110
9330    REM END OF LOOP
9340    RETURN
```

A program that uses this subroutine can now be written. We can use it to look up telephone numbers to replace the serial search in the PHONES program.

We are assuming that the file of names is sorted alphabetically. The subroutine SEARCH should be included after the main program.

You will notice that the binary search program has more instructions than the linear search that it is replacing. Each step is more complicated, but the process is much faster for a large file because fewer steps are executed.

SEARCHING BY ADDRESS CALCULATION

We have seen that the efficiency of the searching process is very much improved by having a file sorted. The next method of searching uses data organized in a way so there is "a place for everything, and everything in its place".

Suppose you had a file of N records numbered from 1 to N. If you knew the number of the record, you would immediately know the location. The number would be the index of the array that holds the file entries. Each entry would have a location where it belonged. The trouble usually is to find the location of a record when what you know is some other piece of information such as a person's name.

Files are sometimes arranged so that they are organized on serial numbers that can be calculated from some other information in the record. For example, we could take a person's name and, by transforming it in a certain definite way, change it into a serial number. This transformation often seems bizarre and meaningless, and we say the name is *hash-coded* into a number. When the number has been determined, the location is then definite and you can go to it without any problem.

Usually with hash coding it happens that several records have the same hash code. This means that, instead of the code providing the address of the exact record you want, what you get is the address of a location capable of containing several different records. We call such a location a *bucket* or *bin.* We then must look at the records in the bin to find the exact one we are interested in. Since the number is small they need not be sorted. A linear search is reasonable when the number of items is small.

If fixed-size bins are used to store the file, it is important to get a hash coding algorithm that will divide the original file so that roughly the same number of records is in each bin.

As an example of a hash-coding algorithm, suppose that we had 1000 bins and wanted to divide a file of 10,000 records into the bins. The file might already have associated with each record an identifying number. For example, it might be a Social Insurance number or a student number. These numbers might range from 1 to 1,000,000. One way to divide the records into bins would be to choose the last three digits of the identifying number as the hash code. Another hash code might be formed by choosing the third, fifth, and seventh digit. The purpose is to try to get a technique that gives about the same number of records in each bin. More complicated hashing algorithms may be necessary.

SORTING

We have already developed a sorting program as an example of step-by-step refinement in Chapter 10. The method we used is called a *bubble sort.* Each pair of neighboring elements in a file is compared and exchanged, to put the element with the larger key in the array location with the higher index. On each exchange pass, the element with the largest key gets moved into the last position. The next pass can then exclude the last position because it is already in order.

We have shown that the binary search technique is much more efficient for a large file than a linear search. In the same way, although a

bubble sort is a reasonable method for a small file, it is not efficient for a large file. What we usually do to sort a large file is to divide it into a number of smaller files. Each small file is sorted by a technique such as the bubble sort, then the sorted smaller files are merged together into larger files.

We will look at an example in which two sorted files are merged into a single larger sorted file.

SORTING BY MERGING

We will develop a subroutine called MERGE to merge a file which has size S1 records ordered on the key field K1$ with a second file, which has size S2 records ordered on the key field K2$ and store it in a third file with key field K3$. Here is the MERGE subroutine:

```
8010    REM SUBROUTINE MERGE(K1$,S1,K2$,S2,K3$)
8020    REM MERGE TWO SORTED FILES
8030    REM LOCAL VARIABLES: I1,I2,I3
8040    REM INITIALIZE LOOP
8050    I1=1
8060    I2=1
8070    I3=1
8080    REM MERGE UNTIL ONE FILE IS USED
8090    REM REPEAT UNTIL I1>S1 OR I2>S2
8100    IF I1>S1 OR I2>S2 THEN GOTO 8230
8110        REM IF K1$(I1)<K2$(I2)THEN...ELSE
8120        IF K1$(I1)<K2$(I2)THEN GOTO 8170
8130            REM DO ELSE
8140            K3$(I3)=K2$(I2)
8150            I2=I2+1
8160            GOTO 8200
8170            REM DO THEN
8180            K3$(I3)=K1$(I1)
8190            I1=I1+1
8200        REM END OF SELECTION
8210        I3=I3+1
8220        GOTO 8100
8230    REM END OF LOOP
8240    REM ADD REMAINING ITEMS TO END OF NEW FILE
8250    REM REPEAT UNTIL I1>S1
8260    IF I1>S1 THEN GOTO 8310
8270        K3$(I3)=K1$(I1)
```

```
8280        I1=I1+1
8290        I3=I3+1
8300        GOTO 8260
8310    REM END OF LOOP
8320    REM REPEAT UNTIL I2>S2
8330    IF I2>S2 THEN GOTO 8380
8340        K3$(I3)=K2$(I2)
8350        I2=I2+1
8360        I3=I3+1
8370        GOTO 8330
8380    REM END OF LOOP
8390    RETURN
```

EFFICIENCY OF SORTING METHODS

The number of comparisons required to merge the two previously sorted files in our example is S1+S2. To sort a file of length N by the bubble sort we can count the maximum number of comparisons that are needed. It is

(N-1)+(N-2)+(N-3)+...+1

This series can be summed and the result is

N(N-1)/2 which is
$N^2/2 - N/2$

When N is large, the number of comparisons is about $N^2/2$, since this is very large compared to N/2. We say the execution time of the algorithm varies as N^2; sorting 100 items takes 100 times the number of comparisons that sorting 10 items does. We will now make calculations to see why sorting by merging is useful for long files. To sort a file of N items, by first using a bubble sort on two files N/2 in length then merging, requires $N^2/4$-N/2 for the bubble sort and N for the merge. This makes a combination total of

$N^2/4 + N/2$ comparisons.

Using a bubble sort on the whole file gives a result of

$N^2/2 - N/2$ comparisons.

When N is 100, the bubble sort merge method requires 2,550 comparisons, the straight bubble sort requires 4,950 comparisons. We can keep dividing files and subfiles, sorting them by merging, with further improvements. In the limit we have a *successive merge* sort that is efficient enough to be used for large files.

CHAPTER 14 SUMMARY

This chapter has presented methods of searching and sorting that are used in computer programs. These methods manipulate *files* of *records*. Each record consists of one or more *fields*.

A search is based on a *key*, such as a person's name, that appears as one field in a record of a file. A *linear search* locates the desired record by starting at the first record and inspecting one record after another until the given key is found. A linear search is slow and should not be used for large files a faster search method, such as binary search, should be used for large files.

A *binary search* requires that the file be ordered according to the key field of the records. An unordered file can be ordered using one of the sorting methods given in this chapter. The binary search inspects the middle record to determine which half of the file contains the desired record. Then the middle record of the correct half is inspected, to determine which quarter of the file contains the desired record, and so on, until the record is located.

If the key is a number that is identical to the index of the desired record then no searching is required, because the key gives the location of the record. Sometimes the key can be manipulated to create a *hash code* that locates a small set of records, called a *bucket*, that includes the desired record.

A file of records can be ordered using the bubble sort. This method repeatedly passes through the file, interchanging adjacent out-of-order records until all records are in order. The bubble sort is slow and should not be used for large files a faster sorting method, such as sorting by merging, should be used for large files.

A file can be sorted by merging in the following manner. First the file is divided into two sub-files and each of the sub-files is sorted by some method, such as the bubble sort. Then, starting with the first records of the two sub-files, the ordered file is created by passing through the sub-files and successively picking the appropriate (smaller key or alphabetically first key) record. If the sub-files are large, they should be sorted by a fast method, such as a merge, instead of by a bubble sort.

CHAPTER 14 EXERCISES

1. Prepare a file of names and addresses, enter the file and order it alphabetically. Now try looking up the address of a friend in your file. We will find this is all more practical once we learn about files on disk.

2. Write a program that maintains a "lost and found" service. First the program reads entries giving found objects and the finders' names and phone numbers. For example, this entry

SIAMESE CAT, MISS MABEL DAVIS, 714-3261

means Miss Mabel Davis, having phone number 714-3261, found a Siamese cat. These entries are to be read and ordered alphabetically and then a similar set of entries for losers of objects is to be processed. If a lost object matches a found object, then the program should display the name of the object as well as the finder, the loser and their telephone numbers. Assume the loser entries are not alphabetized. Process each loser entry as it is read, using a binary search.

3. Test the subroutine SEARCH given in this chapter using a main program BINARY as given here

```
010     REM PROGRAM BINARY
015     REM READ IN FILE
020     INPUT "SIZE="; SIZE
025     PRINT "NAME"; TAB(10); "NUMBER
030     FOR I=1 TO SIZE
040         INPUT FILE$(I),NUMBER$(I)
050         NEXT
055     REM INFINITE LOOP
060         INPUT "NAME="; KEY$
070         REM CALL SUBROUTINE SEARCH
080         GOSUB 9010
090         PRINT "NUMBER IS "; NUMBER$(PLACE)
100         GOTO 60
110     END
   (include subroutine SEARCH here)
```

In testing the program try adding an output instruction to the subroutine so that you can trace what is happening. A suitable one might be

```
9125    PRINT FIRST,LAST,MIDDLE
```

Try putting the file in as a series of data lines in the program and change the instruction in line 40 to be

READ FILE$(I),NUMBER$(I)

Investigate what happens when you ask for a name that is not in the file. Can you change the SEARCH subroutine so that for such a name the output is

NUMBER IS UNLISTED

4. Test the subroutine MERGE given in this chapter by using a main program such as

```
010    REM PROGRAM FILEMERGE
020    INPUT "S1=";S1
030    PRINT "ENTER FILE ONE NAMES"
040    FOR I=1 TO S1
050        INPUT K1$(I)
060        NEXT
070    INPUT "S2=";S2
080    PRINT "ENTER FILE TWO NAMES"
090    FOR I=1 TO S2
100        INPUT K2$(I)
110        NEXT
114    REM CALL SUBROUTINE MERGE
115    GOSUB 8010
120    PRINT "MERGED FILE"
130    FOR I=1 TO S1+S2
140        PRINT K3$(I)
150        NEXT
160    END
```

(include subroutine MERGE here)

Can you modify FILEMERGE so that it can be used to add single items to a file so that a series of items could be merged one-by-one into a file without arranging the items to be added in alphabetic order?

Chapter 15

MAKING SURE THE PROGRAM WORKS

Throughout this book, we have emphasized structured programming techniques; these include step-by-step refinement, programming with careful control of the use of the GOTO statement, choosing good variable names, and so on. These techniques make it easier to write correct programs. We have also given techniques for testing and debugging programs. In this chapter we will collect and expand upon these techniques for making sure a program works.

SOLVING THE RIGHT PROBLEM

The *specifications* for a program tell what the program must do to solve a problem. Before starting to write a program, the programmer needs the detailed specifications for the program. Suppose the problem is to prepare pay checks for the employees of a company there is an entry giving each employee's name and amount of payment. The programmer needs to know the format of the data entry as well as the format for the pay checks. These formats are part of the specifications for the program to prepare pay checks.

Sometimes the program specifications are not completely agreed upon and written down. If an employee's entry indicates an amount of $0.00, this may mean that the employee is on leave and is to receive no pay check. If the programmer does not know the special significance of $0.00 - because the specifications are not complete - he may write a program that prepares hundreds of worthless pay checks. Often programs fail to handle special situations such as $0.00 correctly. If the programmer is in doubt, he should check the specifications and make sure they are complete.

DEFENSIVE PROGRAMMING

Errors are sometimes made in the preparation of data for a program. Amounts may be entered incorrectly; more data may be supplied than anticipated. The method of handling data errors may be given in the program specifications, or it may be left to the discretion of the programmer. Sometimes a programmer can write his program so that it detects and reports bad data. This is called *defensive programming.* Some programs are written to accept absolutely any data after reporting a bad data item, the program ignores the item or attempts to give it a reasonable interpretation. If a program is written assuming no data errors, bad data items may prevent the program from doing its job. It is the programmer's responsibility to make his program sufficiently defensive to solve the problem at hand.

ATTITUDE AND WORK HABITS

The quality of a computer program is determined largely by the attitudes and work habits of the programmer. Some programmers underestimate the programming task. They write programs too quickly, they do not test their programs sufficiently, and they are too willing to believe that their programs are correct.

Most programs, when first written, contain some errors. This is not surprising when you consider the vast number of possible programming errors and the fallibility of every programmer. The programmer should take the attitude that a program is not correct until it is shown to be correct.

One good method of preparing computer programs is to write them using a soft lead pencil. This allows easy corrections and improvements by erasing and replacing lines. If a major change is required, an entire page should be recopied. The program should be entered into the computer only when the programmer feels confident that no more changes are required. This method of program preparation can save the programmer a lot of time. The savings come because it is easy to change a program when it is still on paper and fresh in the programmer's mind. Each later change requires the programmer to relearn the program before he can confidently make modifications. A few minutes of desk-checking a program can save hours of debugging time. The programmer who tries to "do it right the first time" comes out ahead, saving his own time and writing programs with fewer errors.

PROGRAM CORRECTNESS

The most effective way to make sure a program works correctly is to study the program thoroughly. It should be read again and again until the programmer is thoroughly convinced that it is right.

It helps if a second programmer reads and approves the program. Ideally, the second programmer should read the program after its author feels that it is correct, but before it is submitted to the computer. The second reader provides a new point of view and may be able to find typical errors such as incorrect loop initialization. But finding a second reader is not always easy.

This process of studying programs to make sure they are correct can be called "proving program correctness". Sometimes programs are proven correct using a mathematical approach; proving that a program is correct is then similar to proving that a theorem in geometry is true. More often, programs are proven correct by a non-mathematical, common-sense approach. The program is considered to have errors until proven correct.

PROGRAMMING STYLE

A program should be easy to read and understand otherwise the job of studying it to verify its correctness will be hopeless. The programmer should strive for a good *programming style*, remembering that other readers will be in a hurry and will be critical of sloppiness or unnecessary confusion in the program. It commonly happens that as a programmer makes a program clearer and easier to understand, ways to improve or correct the program are discovered.

It takes work to write programs that are easy to read - just as it takes work to write clear English. Good writing requires care and practice. One way of making programs readable and understandable is to give them a simple organization - so the reader can easily learn the relationship among program parts. We have previously presented step-by-step refinement and division into subprograms, often called modular programming, as techniques for designing programs. As well as aiding in the writing of programs, these techniques help make programs easier to read.

USE OF REMARKS AND IDENTIFIERS

One of the rules of good programming style is this: remarks and identifiers should be chosen to help make a program understandable. Remarks should record the programmer's intentions for the parts of the

program. It is a good idea to write remarks as the program is being written.

Better programs require fewer remarks, because the program closely reflects the intentions of the programmer. Programs become more difficult to read if they are cluttered with obvious remarks such as

```
REM INCREASE N BY 1
N=N+1
```

Remarks are usually needed to record:

- *Overall purpose of a program.* What problem the program is to solve. As well, remarks may be used to record the program's author and its date of writing.

- *Purpose of each module.* Similar to the remarks for an overall program.

- *Purpose of a collection of statements.* Such a remark might give the purpose of a loop.

- *Assumptions and restrictions.* At certain points in a program, assumptions and restrictions may apply to variables and the data. For example, one program part may assume that another program part has set NUMBEROFACCOUNTS to a positive number less than 20 to indicate the number of customer accounts.

- *Obscure or unusual statements.* As a rule, such statements should be avoided. If they are required they should be explained.

We have used a standard form of remark to begin every program or subroutine. Adopting such conventions is extremely helpful in making your intentions clear.

Well-chosen identifiers make a program easier to read. Each identifier should record the function of the named object. For example, an array used to save account numbers should be named ACCOUNTNUMBER and not ARRAY. A procedure used to read accounts should be given a name such as FIND and not P1 or MARGARET.

If a variable has a very simple purpose, such as indexing through an array, a one-letter name such as I, J or N may be appropriate. This is because these letters are commonly used for indexing in mathematics. But if the index variable has some additional meaning, such as counting input data entries, a longer name may help the reader.

Avoid abbreviations, such as TBNTR for table entry. Avoid acronyms, such as SAX for sales tax. Unless abbreviations or acronyms are well known to the reader before seeing the program, they impose an extra memorization task that interferes with understanding the program.

Avoid meaningless identifiers such as A, B, C, D and TEMP. A single-letter identifier such as D is sometimes appropriate for a simply-used variable when the name D is relevant, for example, it stands for diameter.

TESTING

After the program has been written and studied to verify its correctness, it should be tested. The purpose of testing is to run the program to demonstrate that it is working properly.

The tests must be chosen with care because only a limited number of them can be run. Consider a program designed to sort any list of 100 names into alphabetic order. Certainly we could not test it exhaustively by trying every possible list of 100 names. We would be testing for years! Rather than exhaustive testing we need to design tests which try every type of situation the program is to handle.

Well-designed tests should point out any errors in the program. Ultimately, testing demonstrates errors better than it demonstrates program correctness. When testing reveals an error, that is, a *bug*, in the program, the programmer is faced with a *debugging* task. We shall present debugging techniques later. Right now, we will give techniques for testing.

The programmer will need to study the program in order to design good tests. The tests should make each statement execute at least once - but this is not enough. Suppose the statement

```
AVERAGE=SUM/COUNT
```

is tested and computes the desired average. This does not demonstrate that all is well; it may be that in some situations COUNT can become zero. If this statement is executed with COUNT set to zero, the statement does not make sense. You will receive an error message of the form:

```
? DIVISION BY ZERO ERROR IN line number
```

So, not only should every statement be executed, but it should be executed for the type of situation it is expected to handle. Care should be taken to:

- *Test end conditions.* See that each loop is executed correctly the first time and last time through. See that indexes to arrays reach their smallest and largest possible values. Pay particular attention to indexes and counters which may take on the value zero. If you use an out-of-bounds index you will receive an error message of the form:

```
? BAD SUBSCRIPT ERROR IN line number
```

- *Test special conditions.* See that data which rarely occurs is handled properly. If the program handles error messages, see that each situation requiring such a message is tested.

Designing tests to exercise all end conditions and special conditions is not easy - but it is worthwhile in terms of program reliability.

The programmer should be able to tell from test results if the program is executing correctly. Sometimes this is easy because the program outputs intermediate results as it progresses. Sometimes the programmer will need to add special output statements to verify that the program is running correctly. These statements can:

Display messages to record the statement being executed. For example, a message might be ACCOUNTS SUBROUTINE ENTERED. If you want to know the line number of each statement that is executed, give the command TRACE before you RUN the program. To run without tracing the execution sequence, you must then give the command NOTRACE.

Output values of variables. This allows the programmer to verify by hand that the values are correct. The best time to output variables is when modules start and when they finish, so the programmer can verify that variables were modified correctly.

Display warnings of violated assumptions. Suppose a subroutine is used to set WHERE to the index of the smallest number in a list of 12 numbers. The assumption that WHERE receives a value from 1 to 12 can be tested by

```
IF (WHERE<1) OR (WHERE>12)THEN
   PRINT " ERROR:WHERE=";WHERE
```

Care must be taken to design appropriate PRINT statements for testing. Too much output will not be read by the programmer too little will not give the programmer sufficient information about the execution of the program.

Ideally, tests should be designed before the program is submitted to the computer. With the program still in mind, the programmer can more easily invent tests that try out every statement. Sometimes a programmer discovers that parts of a program are difficult to test; a slight change in the program may overcome this difficulty. It is best to make these changes when the program is still on paper, before time has been invested in entering the program into the computer. Designing tests requires the programmer to read the program with a new point of view. It sometimes happens that this point of view uncovers errors in the program. The best time to fix these errors is when the program is still on paper.

As programs become larger, it becomes increasingly difficult to test them thoroughly. Large programs can be tested by first testing the modules individually. Then the modules are combined into larger modules and these are tested and so on. The process is called *bottom-up testing.* This method of testing uses specially-written test programs that call the modules with various values of parameters, shared variables and input data.

Whenever a program is modified, it should be retested. All the changed parts should be tested. In addition, it is a good idea to test the entire module containing changes, or even the entire program. The reason is that modifications often require a precise understanding of the surrounding program, and this understanding is sometimes not attained. Very commonly, program modifications introduce errors.

DEBUGGING

A program has bugs (errors) when it fails to solve the problem it is supposed to solve. When a program misbehaves we are faced with the problem of debugging - correcting the error. The program's misbehavior is a symptom of a disease and we must find a cure. Sometimes the symptom is far removed from the source of the problem erroneous statements in one part of a program may set variables' values incorrectly and trigger a series of unpredicted actions by the program. When the symptoms appear via incorrect program output, the program may be executing in a different module. The programmer is left with a few clues: the incorrect output. He has to solve the mystery and cure the disease. Solving these debugging mysteries can take more time than writing the program.

When a program contains a bug, this means that the programmer made at least one mistake. We can categorize programmer errors as follows:

Errors in entering the program. INPUT might be mistyped as INPT. These are typing errors.

Errors in using the programming language. The programmer did not understand a language construct. For example, use of the GOTO in an IF...THEN...ELSE construct may be incorrect.

Errors in writing program parts. Although a particular program part was properly designed, it was not correctly written in Basic. For example, a loop designed to read in account entries might always execute once because of writing the loop's terminating condition incorrectly.

Errors in program design. The program parts and their interactions might be improperly designed. The program designer might forget to provide for the initialization of variables used by some modules. You might overlook the fact that one module, say, DISPLAYACCOUNTS, should be called only after calling another module, say, FINDACCOUNTS.

Solving the wrong problem. The programmer did not understand the nature of the problem to be solved. He may have misunderstood the program specifications. Perhaps the specifications were not correct or complete.

This list of possible errors has proceeded from the least serious to the most disastrous. The first kind of errors, such as typing errors, can be corrected easily once detected. The last type of error, misunderstanding the purpose of the program, may require scrapping the entire program and starting over again.

Some programmers are overly optimistic and immediately conclude that any bugs in their programs are not very serious. Such a programmer is quick to make little changes in his program to try to make the symptoms of the problem disappear. The wise programmer knows that program misbehavior is an indication of sloppiness and that sloppiness leads easily to disastrous errors. Program misbehavior is taken as a sign that the program is sick - a checkup is given by studying it.

The overly optimistic programmer is forever saying, "I just found the last bug." When the wise programmer finds a bug, it is taken as an indication that there may be five more.

Many of the least serious errors, such as misspelled keywords, are automatically pointed out by error messages, because the error results in an illegal Basic program. These errors are usually easy to fix. Some errors are particularly treacherous; they seem to defy attempts to correct them. Here is some advice - some of it repeated from earlier parts of this book - to help you track down treacherous bugs.

Read all error messages. In their hurry to read their program's output, some programmers fail to pay proper attention to error messages. These messages may pinpoint a bug.

Beware of confusion between I and 1. Some people can consistently read X=X+I to mean increase X by one. Errors like this can be found by reading the program character by character - as a computer does! In general, the human tendency to read what we want to be there, rather what is actually there makes debugging difficult.

Beware of misspellings. Some words are easily misspelled. A person who is concentrating on understanding a program may overlook RECEIPT occasionally spelled as RECIEPT. When this happens to involve the first two letters in a variable identifier serious trouble can result.

Beware of language peculiarities. Basic was designed to minimize language peculiarities, but it still has some traps for the naive programmer.

If everything else fails in the debugging effort, the programmer is forced to rerun his program to gain more information about the errors. The programmer may add statements to output variables or to trace the program's execution using TRACE. Extra PRINT statements are designed using the same techniques used in testing to show programs work properly. If the original tests had been carefully enough designed, there is a good chance they would have pinpointed the error and eliminated later time-consuming debugging. As well, it is possible to give direct execution PRINT instructions after a program has run. In this way you can see the final values of any of the program variables. Variables are not set back to zero until another program is loaded or run.

CHAPTER 15 SUMMARY

In this chapter we have listed techniques for making sure a program works. There are a vast number of ways a program can be wrong, so the programmer should learn to be careful at all the stages of program preparation. When a programmer is too hasty to submit his program to the computer, this results in persistent bugs and excessive time spent in debugging. The following important techniques and terminology were presented in this chapter.

Program specifications - explanation of what a program is to do. This should include the forms of the input and output data and the type of calculation or data manipulation to be performed. Essentially, program specifications explain how the computer is to be used to solve a particular problem.

Programming habits - the way a programmer goes about his work. Ideally, you should take the slow but sure approach, completing your program in pencil and thoroughly studying it before entering it into the computer.

Program correctness - studying a program to verify that it satisfies its specifications.

Programming style - if the style is good, then the program can be easily read and understood.

Use of remarks and identifiers - good programming style requires that remarks and identifiers be chosen to make a program understandable. Remarks should record the programmer's intentions; identifiers should record the function or use of the named object.

Testing - running a program to demonstrate that it meets its specifications. Tests should be designed to try every type of situation the program is to handle. Ultimately, testing is better at demonstrating bugs than demonstrating program correctness.

Debugging - correcting errors in a program. Debugging can be the most difficult and time-consuming part of trying to make a program work. These difficulties can be minimized by using the techniques listed in this chapter.

CHAPTER 15 EXERCISES

1. In this exercise you are to use defensive programming. Modify the program given in chapter 9 so that it will handle errors in the data gracefully. The program reads a list of names and outputs the list in reverse order.

2. Try to write a program that is completely correct before you submit it to the computer. Have a friend help you by studying your program for errors after you are convinced that it is free of errors. Record the time you spend preparing the program and record any programming errors you make. Your program should perform one of the following tasks:

(a) The program should read a series of integers followed by the dummy value 9999. Output the sum of the positive integers and the number of negative integers.

(b) The program should read and output a list of alphabetically ordered names. If a name is repeated in the data, it should be output only once.

Chapter 16

FILES AND RECORDS

So far we have spoken about files of records and discussed the process of searching for particular records. This process was made more efficient by having the files sorted. We then looked at ways of sorting files of records. All sorting methods involve moving records around in the computer memory. In our sorting examples, we did not really deal with the situation of sorting records that consisted of more than the one field, namely the key field of the ordering. In our examples, then, moving the record meant only moving this one field. In most data processing applications, records contain a number of fields, and when records are moved each field must be moved separately.

When large quantities of data have to be processed or when you want to keep files over a period of time, it is impossible to store files of records in the main memory of the computer. It is usual to keep files in secondary storage such as magnetic disk storage. We must then be able to read records from such a file and write records into it. We will be looking at the statements in Basic that permit us to manipulate files in secondary storage.

RECORDS

A record is a collection of several fields. As a simple example, suppose that we want to describe each entry in the telephone book as a record. We would identify the three fields as

NAME$
ADDRESS$
NUMBER$

Here is a diagram showing the fields:

The field identifiers are NAME$, ADDRESS$, and NUMBER$. We say the record has a structure, namely three fields. A record structure is sometimes called the *layout* of a record.

To move the record into a new location we need to write assignment statements, one for each field. Temporary locations for holding the fields might be TNAME$, TADDRESS$, and TUMBER$. (If we had used TNUMBER$ instead of TUMBER$ as a variable name it would have been the same as TNAME$ since only the first two letters count.) Arrays of records can be formed by having an array for each field. An array of records can be used for grouping records for sorting purposes. A procedure for sorting a group of records that have been placed in the memory in the main program will be given. The records are to be sorted on the key NUMBER$. The parameters that the subroutine has are COUNT, NAME$, ADDRESS$, and NUMBER$.

```
2010   REM SUBROUTINE SORT(COUNT,NAME$,ADDRESS$,NUMBER$)
2020   REM SORT RECORDS WITH FIELDS
2025   REM NAME$(I),ADDRESS$(I),NUMBER$(I)
2030   REM KEY TO SORT IS NUMBER$(I)
2040   REM LOCAL VARIABLES:TNAME$,TADDRESS$,TUMBER$,I2,J2
2050   FOR I2=1 TO COUNT-1
2060      FOR J2=1 TO COUNT-I2
2070         REM IF NUMBER$(J2)>NUMBER$(J2+1)THEN
2080         IF NUMBER$(J2)>NUMBER$(J2+1)THEN GOTO 2100
2090            GOTO 2210
2100            REM DO THEN
2110            REM SWAP RECORDS J2 AND J2+1
2120            TNAME$=NAME$(J2)
2130            TADDRESS$=ADDRESS$(J2)
2140            TUMBER$=NUMBER$(J2)
2150            NAME$(J2)=NAME$(J2+1)
2160            ADDRESS$(J2)=ADDRESS$(J2+1)
2170            NUMBER$(J2)=NUMBER$(J2+1)
2180            NAME$(J2+1)=TNAME$
2190            ADDRESS$(J2+1)=TADDRESS$
2200            NUMBER$(J2+1)=TUMBER$
2210         REM END OF SELECTION
2220         NEXT
2230      NEXT
2240   RETURN
```

In this example we have an array of records. But we have not yet said how records may be read into or written out from the main memory. The input-output statements the INPUT and PRINT, can be used to read or output individual fields of a record in exactly the same way as the values of individual variables are read or output.

If you input records from the keyboard, each field is read independently. Here is a program that reads a set of at most 25 customer records, sorts them and outputs them:

```
010    REM PROGRAM NUMBERS
020    REM INPUT, SORT BY NUMBER AND OUTPUT RECORDS
030    INPUT "NUMBER OF RECORDS="; COUNT
040    REM INPUT ARRAY OF RECORDS
050    PRINT "ENTER DATA IN QUOTES, SEPARATED BY COMMAS"
060    PRINT "NAME";TAB(16);"ADDRESS";TAB(32);"NUMBER"
070    FOR I=1 TO COUNT
080        INPUT NAME$(I),ADDRESS$(I),NUMBER$(I)
090        NEXT
100    REM SORT RECORDS BY NUMBER
110    GOSUB 2010
120    REM OUTPUT SORTED ARRAY OF RECORDS
130    PRINT
140    PRINT "LIST SORTED BY PHONE NUMBER"
150    PRINT "NUMBER";TAB(10);"NAME";TAB(20);"ADDRESS"
160    FOR I=1 TO COUNT
170        PRINT NUMBER$(I);TAB(10);NAME$(I);
             TAB(20);ADDRESS$(I)
180        NEXT
190    END
```

Here is a sample display:

```
NUMBER OF RECORDS=5
ENTER DATA IN QUOTES, SEPARATED BY COMMAS
NAME                    ADDRESS                         PHONE
? "JOHNSTON,R.L.",      "53 JOHNSTON CRES.",            "491-6405"
? "KEAST,P.",           "77 KREDLE HAVEN DR.",          "439-7216"
? "LIPSON,J.D.",        "15 WEEDWOOD ROAD",             "787-8515"
? "MATHON,R.A.",        "666 REGINA AVE.",              "962-8885"
? "CRAWFORD,C.R.",      "39 TREATHERSON AVE.",          "922-7999"

LIST SORTED BY PHONE NUMBER
PHONE       NAME                ADDRESS
439-7216    KEAST,P.            77 KREDLE HAVEN DR.
491-6405    JOHNSTON,R.L.       53 JONSTON CRES.
787-8515    LIPSON,J.D.         15 WEEDWOOD ROAD
922-7999    CRAWFORD,C.R.       39 TREATHERSON AVE.
962-8885    MATHON,R.A.         666 REGINA AVE.
```

FILES IN SECONDARY MEMORY

In our discussion of files so far, we have had the files stored in the main memory. As soon as we turn the power off in the computer or try to run another program the files will disappear. We must have a way to save files on disk just as we saved programs. Also in most real file applications, the files are too large to be contained in main memory. The part of the file being processed must be brought into main memory, but the complete file is stored in disk memory.

A file in secondary storage is a collection of values all of the same type. The collection is sometimes called a *dataset.* One item at a time may be transferred from the dataset to main memory, or from main memory to the dataset. The item that is transferred must be the *next* item in the sequence of records in the dataset. We say that the file can be read sequentially from the secondary memory to the main memory or written sequentially from the main memory to secondary memory. This kind of file is called a *sequential file.* Another kind of file, called a random-access file, is possible in Applesoft Basic; we will discuss it later.

Files of data are stored on disk in a different way from programs. We call data files *text files.* When the name of a data file (or dataset) appears in the catalogue of files it has the letter T in front of the name. The T stands for text. Our programs, remember, have an A for Applesoft Basic alongside their names in the catalogue of files. The file name of a data file is used to refer to the file within the Basic program.

Disk file handling is not part of ordinary Basic so that the statements that must be used are not simply written as other statements, like INPUT and FOR, are in the program. The disk handling statements must be

1) enclosed in quotation marks
2) preceded by the words PRINT D$; where D$ has been previously assigned the value CHR$(4) which in the ASCII code is a CTRL-D. (Do not try to understand why you do it this way. Just do it.)

The disk handling statements are of the form

OPEN	file name
CLOSE	file name
READ	file name
WRITE	file name
APPEND	file name
DELETE	file name

Before a sequential file can be either read from or written into we must give the statement

OPEN file name

This positions the file reader at the first item of the file. To prepare to read the first item in the file we use the instruction

READ file name

The actual act of reading must be done using an ordinary input instruction of the form

INPUT list of fields in record separated by commas

This instruction follows the READ instruction. If we want, instead, to write on a file we use the disk handling statement (of course enclosed in quotes and preceded by PRINT D$;)

WRITE file name

to prepare to write the next record. The actual writing is done by the ordinary Basic instruction that follows and has the form

PRINT list of items to be written separated by ;",";

This instruction places commas between the items in the disk file. We do *not* read and write on the same sequential file as our information would get all muddled.

When we are finished with a file it must be closed by the disk handling statement

CLOSE name of file

If we are reusing a file that has already had something recorded on it, it is best to wipe it clean, then open it. This is done by the disk handling statements

```
OPEN    file name
DELETE  file name
OPEN    file name
```

If you have been reading from a disk file and want to input some data from the keyboard without closing the file, you can place the statement

PRINT D$

in your program. The INPUT statement that follows this line will cause data to be read from the keyboard. To return to reading from the file you need to execute the

READ file name

disk handling instruction again.

This same method is used when writing to a file. To output on the screen give the statement

```
PRINT D$
```

and the following PRINT statement will be to the screen.

PREPARATION OF FILES

As a first example of the use of the disk handling statements we will show how to prepare a file called CUSTOMERFILE which contains records with the three alphabetic fields: a name field, an address field, and a number field. The file is to be kept sorted with the name field as the key. The last record in the file is to be a dummy record. We will read in a series of records in proper order and store them in the file. The alphabetic information, for example the names, may contain commas so we will have to enclose the different items in quotes. Here is the program that will prepare the file:

```
010    REM PROGRAM INPUTFILE
020    REM READ RECORDS FROM KEYBOARD INTO FILE
030    DUMMY$="ZZZ"
040    D$=CHR$(4):REM CONTROL-D
050    PRINT D$;"OPEN CUSTOMERFILE"
060    REM INPUT FIRST RECORD FROM KEYBOARD
070    PRINT "ENTER DATA IN QUOTES, SEPARATED BY COMMAS"
080    PRINT "NAME";TAB(16);"ADDRESS";TAB(32);"NUMBER"
090    INPUT NAME$,ADDRESS$,NUMBER$
100    REM REPEAT UNTIL NAME$=DUMMY$
110    IF NAME$=DUMMY$ THEN GOTO 190
120        REM WRITE RECORD TO FILE
130        PRINT D$;"WRITE CUSTOMERFILE"
140        PRINT NAME$;",";ADDRESS$;",";NUMBER$
150        REM READ NEXT RECORD
160        PRINT D$
170        INPUT NAME$,ADDRESS$,NUMBER$
180        GOTO 110
190    REM END OF LOOP
200    REM ADD DUMMY RECORD TO FILE
210    PRINT D$; "WRITE CUSTOMERFILE"
220    PRINT NAME$;",";ADDRESS$;",";NUMBER$
230    PRINT D$; "CLOSE CUSTOMERFILE"
240    END
```

Notice that the disk handling WRITE statement of line 130 precedes the PRINT statement of line 140 which writes the record to the file. The state-

ment PRINT D$ of line 160 precedes the INPUT statement of line 170 which reads data from the keyboard.

LISTING OF FILES

A program can be listed by using the Basic command LIST but a data file, which is a text file, must be listed by a Basic program. Here is a program to list the customer file prepared in the previous section.

```
010    REM PROGRAM LISTFILE
020    REM DISPLAY RECORDS FROM FILE
030    DUMMY$="ZZZ"
040    D$=CHR$(4):REM CONTROL-D
050    PRINT "NAME";TAB(16);"ADDRESS";TAB(32);"NUMBER"
060    PRINT D$;"OPEN CUSTOMERFILE"
070    REM READ FIRST RECORD FROM FILE
080    PRINT D$;"READ CUSTOMERFILE"
090    INPUT NAME$,ADDRESS$,NUMBER$
100    REM REPEAT UNTIL NAME$=DUMMY$
110    IF NAME$=DUMMY$ THEN GOTO 190
120        REM DISPLAY RECORD
130        PRINT D$
140        PRINT NAME$;TAB(16);ADDRESS$;TAB(32);NUMBER$
150        REM READ NEXT RECORD FROM FILE
160        PRINT D$;"READ CUSTOMERFILE"
170        INPUT NAME$,ADDRESS$,NUMBER$
180        GOTO 110
190    REM END OF LOOP
200    PRINT D$; "CLOSE CUSTOMERFILE"
210    END
```

MONITORING FILE ACTIONS

Information sent between the computer's memory and the disk by READ and INPUT, or WRITE and PRINT, is not normally displayed on the screen. It is possible to monitor these activities during the debugging phases of program development. As well you can monitor the actual commands. To get the monitor to operate, before you run the program involving disk handling operations, give a command of the form

MON [C][,I][,O]

The square brackets indicate that what is inside them can be omitted, although you cannot omit all three. If the C is present the commands are

monitored; if I, the input goes to the screen as well as to the memory, and if O is present, the output goes to the screen as well as the disk.

To stop the monitoring you must give the command

NOMON [C][,I][,O]

where the options are the same as before. The parameters C, I, and O can be in any order but must be separated by commas. The default option is not to monitor anything.

If you try to use TRACE with a program containing disk handling statements these disk handling statements in the program will not work.

Sometimes, when you are monitoring, the output is displayed on the screen so rapidly that you cannot follow it without stopping it. You can slow down the speed at which characters are sent to the screen of the computer by a command:

SPEED=125

Then the machine will operate at half speed. To restore it to normal speed after you are finished monitoring, give the command

SPEED=255

FILE MAINTENANCE

As a more complicated example of reading and writing files we will program a simple file-maintenance operation. We will assume that there exists on the disk a file of telephone customers records called CUSTOMERFILE and we want to update this file by adding new customers. The information about the new customers is entered on the keyboard. Each input line corresponds to a *transaction* that must be *posted* in the file to produce an up-to-date customer file. This is an example of *file maintenance.* The customer file is ordered alphabetically by the customer's name and the transactions must also be arranged alphabetically. This program will be very similar to the merge-sort program of Chapter 15, except that the records of the two files being merged are not in an array.

```
010    REM PROGRAM UPDATE
020    REM ADD NEW CUSTOMERS TO FILE
030    DUMMY$="ZZZ"
040    D$=CHR$(4):REM CONTROL-D
050    PRINT D$;"OPEN NCUSTOMERFILE"
060    REM READ FIRST RECORD FROM CUSTOMER FILE
070    PRINT D$;"OPEN CUSTOMERFILE"
080    PRINT D$;"READ CUSTOMERFILE"
090    INPUT NAME$,ADDRESS$,NUMBER$
```

```
100    REM READ FIRST TRANSACTION
110    PRINT D$
120    PRINT "ENTER DATA IN QUOTES, SEPARATED BY COMMAS"
130    PRINT "NAME";TAB(16);"ADDRESS";TAB(32);"NUMBER"
140    INPUT TNAME$,TADDRESS$,TUMBER$
150    REM REPEAT UNTIL TNAME$=DUMMY$ AND NAME$=DUMMY$
160    IF TNAME$=DUMMY$ AND NAME$=DUMMY$ THEN GOTO 360
170        REM IF TNAME$>NAME$ THEN...ELSE
180        IF TNAME$>NAME$ THEN GOTO 270
190            REM DO ELSE
200            REM WRITE TRANSACTION TO NEW FILE
210            PRINT D$;"WRITE NCUSTOMERFILE"
220            PRINT TNAME$;",";TADDRESS$;",";TUMBER$
230            REM READ NEXT TRANSACTION
240            PRINT D$
250            INPUT TNAME$,TADDRESS$,TUMBER$
260            GOTO 340
270            REM DO THEN
280            REM WRITE FILERECORD TO NEWFILE
290            PRINT D$;"WRITE NCUSTOMERFILE"
300            PRINT NAME$;",";ADDRESS$;",";NUMBER$
310            REM READ NEXT RECORD
320            PRINT D$;"READ CUSTOMERFILE"
330            INPUT NAME$,ADDRESS$,NUMBER$
340        REM END OF SELECTION
350        GOTO 160
360    REM END OF LOOP
370    REM ADD DUMMY RECORD TO END OF NEWFILE
380    PRINT D$;"WRITE NCUSTOMERFILE"
390    PRINT NAME$;",";ADDRESS$;",";NUMBER$
400    PRINT D$;"CLOSE CUSTOMERFILE"
410    PRINT D$;"CLOSE NCUSTOMERFILE"
420    END
```

Here is the display:

```
NAME                    ADDRESS                 PHONE
(transactions one to a line)
"ZZZ",                  "ZZZ",                  "ZZZ"
```

The dummy value of ZZZ alphabetically follows any legal names in the file. This value is used because in the merging loop, comparisons with the dummy will force all legal names to be merged before the dummy value.

This way of operating, where the transaction data is all ready in sequential order before the program starts to execute is similar to the mode of operating a computer known as *batch processing.* When transactions are entered at random on the keyboard one at a time and the program processes them before the next is entered, we call it *interactive processing.* Interactive processing is particularly appropriate when different responses are requested from the person at the console depending on what the computer does. In a true interactive situation, files are not processed sequentially but instead are accessed at random. To be processed randomly rather than sequentially, a file must be a Basic random-access file.

RANDOM ACCESS TO FILES

In APPLESOFT II Basic it is possible to access disk files randomly. Any record can be read at random and any record may be written at random.

We have said that it is not possible to both read and write from a sequential file or a muddle will result. This is because each record is allotted only as much space as it requires. Short names are given less space than long names. If we want to be able to read and write on the same file at random we must allocate a fixed size space for each record. This space must be big enough to accommodate the longest string of characters that will be stored in the record. This means that a random-access file requires more space on the disk that a sequential-access file.

Random-access files are used if you want to have fast access to individual records as you do in an interactive system. The difference between the two kinds of files is that

(a) The OPEN disk handling statement requires that the length of the record be specified; this is done by a statement of the form

OPEN file name, L length of record in bytes

One byte is required for each character.

(b) Each READ and WRITE disk handling statement must specify a record number R parameter; this is done by a Basic statement of the form

PRINT D$ "READ file name, R"; record number

In any disk file of records the records have an implicit address. The first record has address 0; it has *record number* 0. The second has record number 1, and so on. To position the read/write heads at a particular record you use the Basic statement

```
PRINT D$;" READ file name, R"; record number
```

Once you are in position you may read the record by an INPUT instruction.

The question is: How do you know what the record number of a particular record in the file is in order to seek it? One way is to keep a directory for the file, like a telephone directory, which will give you the record number of any record you want. We would then search the directory for the record number by any of the means we have discussed for searching and then go to the complete record. As new records are added to the file, new directory items are added to the directory. The directory may be kept sorted in order to speed up the search, using binary search, for example. But the file itself need not then be kept sorted. A search through a directory can be a lot faster than a search through a file because the directory can be kept in the high speed memory during the search. The directory can of course be stored as a separate, single-record file on the disk.

SYSTEM COMMANDS FOR TEXT FILES

We have said that text files are listed in the catalogue with a T in front of them. We *cannot* create these files by typing them in and using the SAVE command as we did program files. We have to read them in, through a Basic program, and write them to a text file. Naturally we cannot RUN a text file.

There are, however, certain system commands that can be used with text files. These are LOCK and UNLOCK, DELETE, and RENAME.

For instance, in the file maintenance program we prepared an updated (new) customer file called NCUSTOMERFILE. When we perform the updating we expect the current customer information to be stored in CUSTOMERFILE. We could give the system commands

```
DELETE CUSTOMERFILE
RENAME NCUSTOMERFILE, CUSTOMERFILE
```

to delete the old customer file and rename the updated file CUSTOMERFILE so that the UPDATE program could be run for the next updating. It is a good idea to DELETE the earlier version of the customer file so that no confusion could result in the files.

Often people like to keep a backup copy of a file that is going to be deleted or have a new version take its place. You can do this by preceding the other commands by the command

```
RENAME CUSTOMERFILE, BUCUSTOMERFILE
```

The file used to back up (BU) the process can be deleted when the next generation of customer file appears.

TEXT FILES FOR EXECUTION

We have said that, naturally enough, you cannot RUN a text file. If the text file is data this is nonsense anyway. But text files can be files of system commands. If they are, the way that we run them is to use the system command:

EXEC text file name

Here is an example of a program to prepare a text file that can be executed with EXEC.

```
010    REM PROGRAM PREPARE
020    REM PRODUCES A FILE CALLED CHANGE
030    D$=CHR$(4): REM CONTROL-D
040    PRINT D$;"OPEN CHANGE"
050    PRINT D$;"WRITE CHANGE"
060    PRINT "DELETE BUCUSTOMERFILE"
070    PRINT "RENAME CUSTOMERFILE, BUCUSTOMERFILE"
080    PRINT "RENAME NCUSTOMERFILE, CUSTOMERFILE"
090    PRINT "RUN LISTFILE"
100    PRINT D$; "CLOSE CHANGE"
110    END
```

When we give the command

RUN

the text file called CHANGE will have written in it

```
DELETE BUCUSTOMERFILE
RENAME CUSTOMERFILE, BUCUSTOMERFILE
RENAME NCUSTOMERFILE, CUSTOMERFILE
RUN LISTFILE
```

We are assuming for this example that we have the files BUCUSTOMERFILE, CUSTOMERFILE and NCUSTOMERFILE and the program LISTFILE, shown in this chapter, stored on the diskette.

To cause the text file CHANGE to be executed we give the command

EXEC CHANGE

This causes the present CUSTOMERFILE to be stored as BUCUSTOMERFILE and the file NCUSTOMERFILE, prepared by the program UPDATE, to be stored as CUSTOMERFILE then listed by LISTFILE. It is important that the original file BUCUSTOMERFILE be deleted before any other file is renamed as BUCUSTOMERFILE as two files of the same name would result.

In this chapter we have shown programs for creating files, listing files, and updating files. We could run these programs one after the other by giving a series of RUN commands. The series of programs could be joined in a chain so that when one is finished the next would be run. In Applesoft Basic as each new program is started the previous program and all its variables are erased. This is fine as far as chaining these programs goes since what we want to save from one to the next is safely on the disk. The disk operating system manual (DOS 3.2) describes what is necessary to chain programs when you want to save the values of variables from one to the next.

Programs can be chained to run one after the other by including, as the last executable statement in the first program to be run, a command to run the second, and so on. To include a command that refers to a disk file in a program it must, as other disk-handling instructions are, be enclosed in quotes and preceded by PRINT D$;. To run the UPDATE program after the INPUTFILE program we could include, just before the END of the INPUTFILE program the instruction

```
PRINT D$;"RUN UPDATE"
```

System commands such as LIST that do not involve reference to the disk may be incorporated directly in a program without enclosing them in quotes and prefacing them by PRINT D$;. For example, here is a program that lists itself.

```
010     REM PROGRAM SELFLIST
020     REM THIS PROGRAM LISTS ITSELF
030     LIST
040     END
```

When the program is run a listing of the program appears on the screen.

SAVING PROGRAMS AS TEXT FILES

When the command SAVE is given for a program, it is stored as an A-type file (the A standing for Applesoft Basic). We can also, by a rather clever trick, capture a program listing as a text file. If such a file is "executed" by EXEC the result is to read the program listing into the memory

without disturbing what is already in the memory. When we LOAD a program into memory whatever is there is erased. This process of capturing programs can be helpful in incorporating subroutines in programs.

What you have to do is have a special little program we will call CAPTURE in the memory along with the program or subroutine that you want to save in a text file. Here is a CAPTURE program.

```
1       REM PROGRAM CAPTURE
2       D$=CHR$(4)
3       PRINT D$;"OPEN CAPTURE"
4       PRINT D$;"WRITE CAPTURE"
5       POKE 33,30
6       LIST 1-9
7       PRINT D$:"CLOSE CAPTURE"
8       TEXT
9       END
```

Type in this program then type RUN. You will notice the disk operating but nothing appears on the screen.

Now try typing NEW then LIST; nothing is in memory. Now type

```
EXEC CAPTURE
```

You will see a series of 9 prompt symbols (]), one on the left of each of 9 lines. You have by this move read the CAPTURE program into memory. Try typing LIST and you will see.

In line 6 of the CAPTURE program the command LIST 1-9 places the lines 1 to 9 on the disk file called CAPTURE. Try typing CATALOG and you will see that there is a text file (labelled with a T) called CAPTURE. The instruction in line 5 (POKE 33,30) is one that places the value 30 in machine address 33. This is where the Basic interpreter looks to find the width of the line (in characters) on the display screen. A width of 30 keeps the listing that you will record in the text file the right size. Line 8 makes sure the screen is back in text mode with the full 40 character width as you finish.

We have shown how the CAPTURE program can save itself as a text file. Now we will use a modification of the CAPTURE program to save any other program or subroutine. Suppose we had a program called PROG that we have entered into the computer. It runs from line 010 to line 350. When we have saved the program in the usual way using SAVE we can, as well, save it as a text file. To do this type

```
EXEC CAPTURE
```

Now type LIST and you will see that the CAPTURE program has been included as lines 1 to 9 along with your program. You must now customize the CAPTURE program to save *your* program. This is done by editing lines 3, 4, and 7 by changing the word CAPTURE to TPROG (the name you are giving to the text file version of PROG). Now change line 6 to

```
LIST 010-350
```

so that CAPTURE will just capture your program and not itself this time. These changes can probably be done most easily using the cursor movement.

Now you are ready to type RUN. Since any program starts to run at the lowest line number only the CAPTURE program is run. It stops when the END is reached in line 9.

To prove that you have saved PROG as a text file called TPROG, type CATALOG. It should be in the catalogue. (When the number of files on the disk is larger than what can be displayed on the screen at one time, the catalogue listing will stop when the screen is full and you must press a key such as return to see the next screenfull.) Now type NEW to be sure nothing is in the memory; then type

```
EXEC TPROG
```

followed by LIST. You should see that your program is there. If you have, as well, by this method captured a subroutine called SUBR with lines numbered from 3010 to 3210 in text file TSUBR then after you type EXEC TPROG type EXEC TSUBR. Now LIST will show that both the program PROG and the subroutine SUBR are there.

This method of storing subroutines as text files can be useful if renumbering of the lines is necessary.

USE OF END-OF-FILE ERROR

We have included in our files on the disk a dummy record which tells us when to stop reading from the file. If you tried to read another record beyond the dummy record an error would occur. We can use this end-of-file error to control our reading of files but we believe that it is a more complicated method than the one that uses the dummy record. Suppose in the program LISTFILE we were printing a file with no dummy record at the end. Here is what our program might be

```
010     REM PROGRAM LISTFILE2
020     REM DISPLAY RECORDS FROM FILE
030     ONERR GOTO 170
040     D$=CHR$(4); REM CONTROL-D
```

```
050     PRINT "NAME";TAB(16);"ADDRESS";TAB(32);"NUMBER"
060     PRINT D$;"OPEN CUSTOMERFILE"
070     REM LOOP UNTIL END OF FILE ERROR
080         REM READ NEXT RECORD
090         PRINT D$;"READ CUSTOMERFILE"
100         INPUT NAME$,ADDRESS$,NUMBER$
110         REM DISPLAY NEXT RECORD
120         PRINT D$
130         PRINT NAME$;TAB(16);ADDRESS$;TAB(32);NUMBER$
140         GOTO 080
150     REM END OF LOOP
155     PRINT D$;"CLOSE CUSTOMERFILE"
160     END

170     REM SET    TO ERROR TYPE
180     CODE=PEEK(222)
190     REM END OF FILE IS CODE 5
200     IF CODE=5 THEN GOTO 150
210     PRINT "ERROR OTHER THAN END OF FILE CODE=";CODE
220     STOP
```

A list of all the error codes is in Appendix 5. In line 030 the instruction

```
ONERR GOTO 170
```

is not executed until there is an interruption due to an error condition. If we try to read a record beyond the last record an error occurs and, if there is not an ONERR GOTO instruction in the program, an error message would be output and the execution would be stopped. In our program when the error occurs we go to line 170 where we look to see what kind of error has occurred. The code for an error is stored in machine location 222 so the value of PEEK(222) is the code number. Error number 5 is an end of data error.

If our program did not include this ONERR GOTO instruction we would not have closed the customer file.

If several files are being read, an error might occur in several different places. We can find out when we arrive in the error handling part of the program at which line of the program the error occurred. The expression

```
PEEK(218) + PEEK(219) * 256
```

gives the line number where the error occurred provided an ONERR GOTO instruction has been executed.

If you want to continue, after an error, exactly where the error occurred you can use the statement

RESUME

in the error handling part of the program. Control will return to the statement which generated the error.

When files are being used without a dummy record it is possible to write additional records on the file without recopying the file. This might be useful when there is no order to what is stored on the file. To do it use the disk handling statement

APPEND file name

followed by

WRITE file name

instead of

OPEN file name

CHAPTER 16 SUMMARY

In this chapter we introduced programming language constructs for manipulating records and using files in secondary memory. The following important terms were discussed in this chapter:

Record - a collection of fields of information. For example, a record might be composed of a name field, an address field and a telephone number field.

Array of records - to declare an array of records of the customer record type we use the declaration:

DIM NAME$(100),ADDRESS$(100),NUMBER$(100)

This creates an array of 100 records where each record has the same fields; NAME$(5),ADDRESS$(5), and NUMBER$(5) refer to the name, address, and number fields of the fifth record in the array.

Dataset - a file of information residing on secondary storage, typically on a disk.

Sequential files - files that are always accessed (read or written) in order, from first item to second item to third item and so on.

Random-access files - files in which any particular record is accessed directly by means of its record number. The records of a file are numbered sequentially starting from 0 for the first record.

Disk handling statements - instructions that can be imbedded in a Basic program to enable you to read and write information from or to a disk. The disk handling statements must be enclosed in quotes and preceded by PRINT D$; where the variable D$ has already been assigned the value CHR$(4). Disk handling statements in a Basic program have the form:

```
PRINT D$; "disk handling statement"
```

and the assignment

```
D$=CHR$(4)
```

must appear at the beginning of the program.

OPEN a file - means to prepare a file for reading or writing by a program. A file in Basic can be opened using the disk handling statement:

```
OPEN file name
```

When OPEN is executed for a sequential file the file is positioned to its first item (if any).

PRINT D$ - the statement that you place in a Basic program in which you are using disk handling statements if you want the following INPUT statement to take data from the keyboard or the following PRINT statement to display data on the screen.

READ - prepare a file to be read; the INPUT statements that follow take their information from the file rather than the keyboard. The disk handling statement has the form

```
READ name of file
```

WRITE - prepares to add a record to a file. The disk handling statement has the form:

```
WRITE file name
```

CLOSE - files that have been opened by OPEN must be closed after processing is finished by the disk handling statement

```
CLOSE file name
```

MON - system command used in the form

```
MON [C][,O][,I]
```

where if C is present disk commands are monitored; if O is present output to the disk is displayed on the screen; and if I is present the input from the disk is displayed on the screen.

NOMON - a system command used to stop monitoring of disk activity. The form is the same as for MON.

SPEED=x - a command that sets the speed at which the computer sends characters to the screen. The slowest is x=0; the fastest is x=255. With this command you can produce a slow motion display for use during monitoring.

File maintenance - means to keep a file up to date. This involves reading transactions and adding, deleting or modifying file records.

Text file - a file of data, as opposed to a program that is filed on the disk. Text files have the letter T in their catalogue listing. Text files cannot be handled by SAVE or LOAD and naturally do not RUN. Programs and subroutines can be captured as text files by a special short program.

READ a record of a random-access file - means to find a particular record in the file. The Basic statement

PRINT D$;"READ file name, R"; record number

will position the read/write head of the disk at the required record. After that the record is read by INPUT statements.

EXEC - a command that permits a text file to be executed. If the text file is a list of system commands then the commands are executed. If the text file is a Basic program then the program is placed into memory without disturbing what else is there. A series of EXECs can be used to concatenate a program and subroutines.

Commands for text files - the commands DELETE, RENAME, LOCK, and UNLOCK can be used with text files. LOAD, SAVE, and RUN cannot be used.

Chaining programs - causing the execution of one program after another automatically. A command to run the second program is included as the last statement of the first program, and so on. The RUN command must be enclosed in quotes and prefaced by PRINT D$;.

Catalogue listing - when the list in the catalogue is longer than 18 files, the display stops; the next 18 or fewer file names are displayed if you press a key such as return.

Capturing a program or subroutine as a text file - is done by a special short program which resides in memory with the program or subroutine to be captured.

End-of-file error - the error that occurs when an attempt is made to read beyond the last record of a text file. The error code for this is 5. The error code is stored in machine location 222.

ONERR GOTO - a statement that is an error control statement; it should be placed at the beginning of any program whose execution you do not want to stop if an error occurs. The line number following the GOTO is that of a part of the program that will deal with the error and then send control back to the program at an appropriate line.

RESUME - a statement that is placed in an error handling part of the program and which returns control to the line where the error occurred.

PEEK(222) - a function whose value is the code number of the particular error that has occurred. Appendix 5 gives a list of errors and their codes.

PEEK(218)+PEEK(219)*256 - an expression whose value is the line number at which an error has occurred. This is only set if an ONERR GOTO statement has been executed.

APPEND file name - the disk handling statement that is used, instead of OPEN, before a WRITE when you want the writing to be added to what is already on the file. Not used if the file has a dummy record at its end.

CHAPTER 16 EXERCISES

The exercises for this chapter are based on a data processing system to be used by Apex Plumbing Supplies. For each of its customers, Apex has a record with the fields:

NAME$
ADDRESS$
BALANCE
CRLIMIT

These records are entered from the keyboard. However, they are to be transferred to a disk file. A sample of the data is shown:

ABBOT PLUMBING,	94 N.ELM,	3116,	50000,
DURABLE FIXIT,	247 FOREST HILL,	0,	10000,
ERICO PLUMBING,	54 GORMLEY,	9614,	5000,

...

The exercises for this chapter require you to write programs for various parts of the data processing system for Apex.

1. Write a program to read data and create a master file for Apex Plumbing Supplies.

2. Write a program to read an old version of the master file for Apex, sort the file, creating a new master file that is in alphabetic order according to the name field.

3. Write a program that takes an existing master file for Apex and creates a new master file by deleting or adding new customer records. For example, the transaction data entries for your program might be

```
DAVIS REPAIR,        4361 MAIN,        2511,        10000
ERICO PLUMBING,      DELETE,           0,           0
```

You can assume that the transactions are in alphabetic order. If the address field of the transaction specifies DELETE, the account is to be deleted from the file otherwise it is an addition.

4. Write a program that reads the Apex master file and outputs the list of customers whose balances exceed their credit limits.

5. Write a program that reads the Apex master file and outputs a bill for each customer whose balance is greater than zero. For example, for the file record

```
DAVIS REPAIR          4361 MAIN          2511    10000
```

your program should output

```
TO: DAVIS REPAIR
  4361 MAIN

DEAR SIR OR MADAM:
    PLEASE REMIT 25.11 FOR PLUMBING SUPPLIES.
        THANK YOU,

        JOHN APEX, PRES.
        APEX PLUMBING SUPPLIES
        416 COLLEGE ST.
```

6. Write a program that updates the master file using billing and payment transactions. A billing transaction is an entry of the form

```
name
amount
```

For each billing transaction, the balance of the account is to be increased by the specified amount. A payment transaction is an entry of the form

name
amount
CR

For each payment transaction, the balance of the account is to be decreased by the specified amount. The billing and payment entries are in order.

7. Organize the master file so that its records may be accessed randomly by creating a directory file for it. Repeat all the other exercises using random access.

Chapter 17

DATA STRUCTURES

Variables and arrays of variables are examples of what we generally call *data structures.* Just as we systematize our programs by attempting to write well-structured programs, we systematize the way in which data is stored. We structure data.

In this chapter we will describe other structural forms for data and give examples of how these structures are useful to us. We will describe data structures called *linked lists* and *tree structures.* There are many kinds of lists, for example, *stacks, queues, doubly linked lists*, and so on. Tree structures can be limited to *binary trees*, or may be more general.

LINKED LISTS

Suppose that we had a file of records stored in an array called NAME$. The records are arranged in sequence on some key. For simplicity, we will consider that each record consists only of a single field which is the key to the ordering. We know that if the order is ascending and no two keys are identical, then

NAME$(I+1) > NAME$(I)

The difficulty with this kind of data structure for a file comes when a new item is to be added to the file; it must be inserted between two items. This means we would have to move all the items with a key higher than the one to be inserted, one location on in the array. For example, you can see what happens when we insert the word DOG in this list:

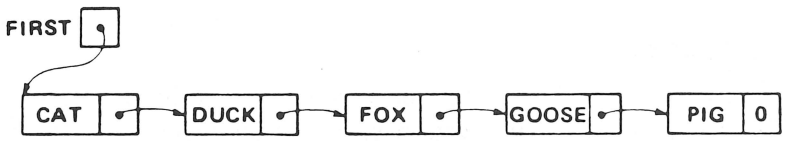
	before	after inserting DOG
NAME$(1)	CAT	CAT
NAME$(2)	DUCK	DOG
NAME$(3)	FOX	DUCK
NAME$(4)	GOOSE	FOX
NAME$(5)	PIG	GOOSE
NAME$(6)	-	PIG

Any list that is changing with time will have additions and deletions made to it. A deletion will create a hole unless entries are moved to fill the hole.

When the list changes with time we can use the data structure called the *linked list.* In the linked list each item has two components, the data component and the linking component or *link.* We associate with each entry in the NAME$ array an entry in a second array of integers called LINK. The number stored in LINK(I) is the index of the next entry in the sequence of the NAME$ array. This means that the actual or *physical sequence* in the NAME$ array is different from the *logical sequence* in the list. Here is an example showing our previous list as a linked list. The start of the list is stored in the variable FIRST.

	FIRST	3	
NAME$(1)	PIG	LINK(1)	0
NAME$(2)	FOX	LINK(2)	4
NAME$(3)	CAT	LINK(3)	5
NAME$(4)	GOOSE	LINK(4)	1
NAME$(5)	DUCK	LINK(5)	2
NAME$(6)	-	LINK(6)	-

Here is a diagram of this:



You can follow the list by beginning with the value of FIRST, which is 3. The first entry will be in NAME$(3); it is CAT. By looking then at LINK(3) you find a 5 which is the index of the next list item, NAME$(5), which is DUCK. You follow the list down until you reach a LINK whose value is 0; this is the signal that you have reached the end of the list. Other signals can be used, such as having a negative number.

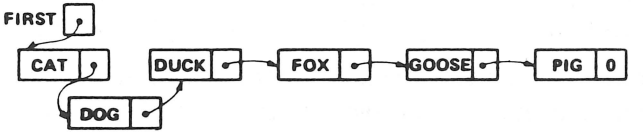
INSERTING INTO A LINKED LIST

To see the merit of a linked list we must see how to insert new entries. We will add DOG in its proper list position. We will do this first by hand; afterwards we will have to program it for the computer. We will place DOG in NAME$(6) since it is an available or free location. We must now change the values of certain of the links so that the new entry will be inserted. We must put a value into LINK(6) and change the value of the LINK of the entry before DOG, which is CAT, to point to NAME$(6).

This means that LINK(3) must be changed to 6 and LINK(6) must be set to 5 so that the entry after DOG is DUCK, which is NAME$(5). The linked list then becomes:

	FIRST	3	
NAME$(1)	PIG	LINK(1)	0
NAME$(2)	FOX	LINK(2)	4
NAME$(3)	CAT	LINK(3)	6
NAME$(4)	GOOSE	LINK(4)	1
NAME$(5)	DUCK	LINK(5)	2
NAME$(6)	DOG	LINK(6)	5

Here is a diagram:



To add DOG, one LINK must be changed and one set. No movement of the existing items in NAME$ is necessary. This is surely an improvement over moving half the list, on the average, to insert a new entry. The cost of this improved efficiency of operation comes in having to reserve memory space for the LINK array. This array gives the structure of the list and is stored explicitly for a linked list. In an array, the sequence or structure is implicit; each entry follows its neighbor. We will use other kinds of structures that require us to store the structure information explicitly.

MEMORY MANAGEMENT WITH LISTS

With linked lists, some of the memory is used for structure information and some for data. For any list, as the list grows, we use more memory; as it shrinks, we use less. This means we must reserve enough memory to hold the longest list that we ever expect to have. But we should not waste memory. As we stop using certain elements of the array by deleting entries, we must keep track of where they are, so when additions occur we can reuse these same elements. To keep track of the available array elements we keep them together in a second linked list. The list of available array elements does not have any useful information in the NAME$ part, but it is structured as a list using values in the LINK part. We must keep track of the beginning of this list so we keep the index of its beginning in variable AVAILABLE.

Here is an array of 10 elements that stores our previous data items in a different set of locations and has the available space linked up:

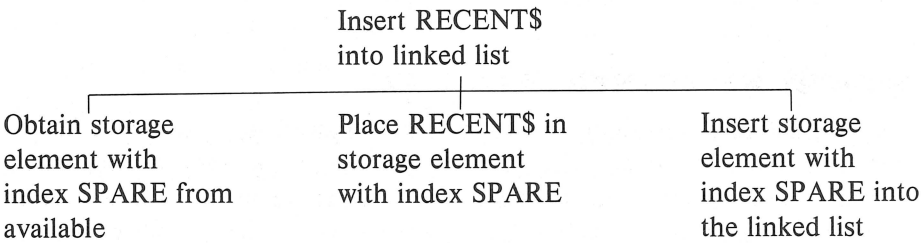
	FIRST 10	AVAILABLE 7
NAME$(1)	GOOSE	LINK(1) 9
NAME$(2)	FOX	LINK(2) 1
NAME$(3)	-	LINK(3) 6
NAME$(4)	DUCK	LINK(4) 2
NAME$(5)	-	LINK(5) 0
NAME$(6)	-	LINK(6) 5
NAME$(7)	-	LINK(7) 3
NAME$(8)	DOG	LINK(8) 4
NAME$(9)	PIG	LINK(9) 0
NAME$(10)	CAT	LINK(10) 8

In these arrays there are two linked lists, one containing the actual data, the other containing elements available for use. Each list has a pointer to its start; each has a last element with a link of 0. Every element of the array is in one list or the other.

The next problem is to write a procedure for adding a new item to the list. We will develop the algorithm for this using step-by-step refinement.

SUBROUTINE FOR INSERTING INTO A LINKED LIST

The first step is to construct a solution tree. We will presume the value to be added is in the variable RECENT$:



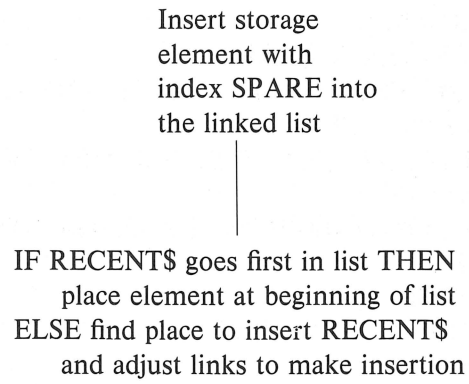
The expansion of the left branch of the solution tree requires us to find the index SPARE of the first element of the list of available elements and remove the element from the list. Here is the program segment that does this:

```
SPARE=AVAILABLE
AVAILABLE=LINK(AVAILABLE)
```

The middle branch is also simple. It is

NAME$(SPARE)=RECENT$

We must expand the right branch still further:



RECENT$ will go first in the list if either the list is empty or RECENT$ is less than the first element of the list. So we can write, "IF RECENT$ goes first in list," in this way:

IF (FIRST=NULL) OR (RECENT$<NAME$(FIRST)) THEN

We have assumed that NULL is a named constant whose value is zero. We can write, "Place element at beginning of list," in this way:

```
LINK(SPARE)=FIRST
FIRST=SPARE
```

For the part of the program after the ELSE we need to examine the entries in the list and compare them with RECENT$. The index of the element being compared we will call AFTER. The index of the previously compared element we will call BEHIND. We need to keep track of this previous element, because if

RECENT$ < NAME$(AFTER)

we must insert our element with index SPARE between BEHIND and AFTER. The whole subroutine can now be written out. We are presuming that NAME$, LINK, FIRST, and AVAILABLE are established in the main program.

```
6010  REM SUBROUTINE INSERT(RECENT$,NAME$,LINK,
        FIRST,AVAILABLE)
6020  REM INSERTS RECENT ENTRY INTO LINKED LIST
6030  REM LOCAL VARIABLES: SPARE,BEHIND,AFTER
6040  REM OBTAIN STORAGE ELEMENTS FROM AVAILABLE
6050  SPARE=AVAILABLE
6060  AVAILABLE=LINK(AVAILABLE)
6070  REM PLACE RECENT ENTRY IN ELEMENT WITH INDEX SPARE
6080  NAME$(SPARE)=RECENT$
6090  REM SEE IF RECENT SHOULD BE FIRST IN LIST
6100  REM IF...THEN...ELSE
6110  IF(FIRST=NULL)OR(RECENT$<NAME$(FIRST))THEN
        GOTO 6260
6120    REM DO ELSE
6130    REM FIND PLACE TO INSERT RECENT
6140    BEHIND=FIRST
6150    AFTER=LINK(FIRST)
6160    REM REPEAT UNTIL(AFTER=NULL)OR
          (RECENT$<NAME$(AFTER))
6170    IF(AFTER=NULL)OR(RECENT$<NAME$(AFTER))THEN
            GOTO 6210
6180      BEHIND=AFTER
6190      AFTER=LINK(AFTER)
6200      GOTO 6170
6210    REM END OF LOOP
6220    REM ADJUST LINKS TO MAKE INSERTION
6230    LINK(BEHIND)=SPARE
6240    LINK(SPARE)=AFTER
6250    GOTO 6290
6260    REM DO THEN
6270    LINK(SPARE)=FIRST
6280    FIRST=SPARE
6290  REM END OF SELECTION
6300  RETURN
```

So far we have ignored a problem in our INSERT subroutine. The conditions for both the IF statement and the REPEAT UNTIL loop use an element of the NAME$ array which may have the index NULL (0). The Basic NAME$ array will automatically have a lower bound of zero. NAME$(0) should be given a value, although the value is not actually part of the list.

DELETING FROM A LINKED LIST

The process of deletion is very similar. We will just record the complete procedure.

```
7010   REM SUBROUTINE DELETE(DISCARD$,NAME$,LINK,
         FIRST,AVAILABLE)
7020   REM DELETE DISCARD FROM LINKED LIST
7030   REM LOCAL VARIABLES: EXTRA,BEHIND
7040   REM FIND WHERE DISCARD IS LOCATED
7050   EXTRA=FIRST
7060   REM REPEAT UNTIL NAME$(EXTRA)=DISCARD$
7070   IF NAME$(EXTRA)=DISCARD$ THEN GOTO 7110
7080       BEHIND=EXTRA
7090       EXTRA=LINK(EXTRA)
7100       GOTO 7070
7110   REM END OF LOOP
7120   REM REMOVE DISCARD FROM LIST
7130   REM IF FIRST=EXTR THEN...ELSE
7140   IF FIRST=EXTR THEN GOTO 7180
7150       REM DO ELSE
7160       LINK(BEHIND)=LINK(EXTRA)
7170       GOTO 7200
7180       REM DO THEN
7190       FIRST=LINK(EXTRA)
7200   REM END OF SELECTION
7210   REM ADD STORAGE ELEMENT TO AVAILABLE
7220   LINK(EXTRA)=AVAILABLE
7230   AVAILABLE=EXTRA
7240   RETURN
```

(Note that in lines 7130 and 7140 the variable EXTRA was abbreviated to EXTR. This makes no difference since only the first two letters matter. It was done because the A on the end of EXTRA in front of THEN produced the keyword AT and a syntax error was reported. Just another Basic problem.)

Before using these two subroutines we must set FIRST to NULL, set AVAILABLE to 1, and LINK(I) to I+1, with the exception of the last element which should have a NULL link.

MAINTAINING A LINKED LIST

Here is a main program that allows you to insert or delete names from a linked list or write it out depending on which of three choices you make

```
010    REM PROGRAM MAINTAIN
020    REM PERMITS INSERTION,DELETION, AND LISTING
030    DIM NAME$(50)
040    DIM LINK(50)
050    NULL=0
060    FIRST=NULL
070    LINK(50)=NULL
080    FOR I=1 TO 49
090        LINK(I)=I+1
100        NEXT
110    AVAILABLE=1
120    NAME$(0)="ZZZ"
130    REM FIND OUT WHAT ACTION IS WANTED
140    INPUT "ENTER (1)INSERT(2)DELETE(3)OUTPUT";TYPE
145    REM CASE CONSTRUCT
150    ON TYPE GOTO 160, 210, 260
160        REM INSERTION
170        INPUT "NAME TO BE INSERTED=";RECENT$
180        REM CALL SUBROUTINE INSERT
190        GOSUB 6010
200        GOTO 130
210        REM DELETION
220        INPUT "NAME TO BE DELETED=";DISCARD$
230        REM CALL SUBROUTINE DELETE
240        GOSUB 7010
250        GOTO 130
260        REM LISTING
270        REM CALL SUBROUTINE OUTPUT
280        GOSUB 5010
290        GOTO 130
295    REM END OF CASE
300    END

5010   REM SUBROUTINE OUTPUT
5020   REM DISPLAY LINKED LIST IN ORDER
5030   I=FIRST
5040   REM REPEAT UNTIL LINK(I)=NULL
5050   IF LINK(I)=NULL THEN GOTO 5090
```

```
5060        PRINT NAME$(I)
5070        I=LINK(I)
5080        GOTO 5050
5090    REM END OF LOOP
5100    RETURN
```

(INSERT and DELETE subroutines as before)

STACKS

In the preceding sections we showed how to insert and delete items for a linked list. The insertions and deletions could be anywhere in the list. In each case, as the list of data items was changed, a second linked list of available storage elements was maintained. A deletion from the list of data items resulted in an addition to the list of available elements; an addition in the data list produced a deletion in the available list. The actions involving the available storage list were much simpler. This is because the additions and deletions for it always were to the beginning of that list. A list that is restricted to having entries to or removals from the beginning only is called a *stack.* The situation is similar to a stack of trays in a cafeteria. When you want a tray you take it off the top of the stack when you are through with a tray you put it back on the top. When a list is used as a stack, we often call the pointer to the beginning of the list UPPER. When an entry is removed from the top we say we have *popped* an entry off. UPPER must then be adjusted to point at the next entry. When we add an entry we say we have *pushed* it onto the stack.

Because a stack change only occurs at one end, it is convenient to implement a stack without using a linked list; an ordinary array will do. In our examples, a linked list is necessary for our stack of available storage elements because they are scattered all over. Stacks have other uses so we will show how a stack can be implemented using an array. We will call the array STACK$. The bottom of the stack will be in STACK$(1), the next entry in STACK$(2), and so on. Sorry if our stack seems to be upside down! Here is a stack of symbols:

```
UPPER       4
STACK$(1)   +
STACK$(2)   -
STACK$(3)   +
STACK$(4)   /
```

This sort of stack is often used by interpreters for translating arithmetic expressions into machine language.

Before using the stack we initialize it to be empty by setting UPPER to zero:

```
UPPER=0
```

To add an item to the stack we can call the subroutine ADD:

```
4010  REM SUBROUTINE ADD(SYMBOL$,STACK$,UPPER)
4020  REM PUSH SYMBOL ONTO STACK
4030  UPPER=UPPER+1
4040  STACK$(UPPER)=SYMBOL$
4050  RETURN
```

To remove the top item from the stack we can call the subroutine SUBTRACT:

```
4100  REM SUBROUTINE SUBTRACT(SYMBOL$,STACK$,UPPER)
4110  REM POP SYMBOL OFF STACK
4120  SYMBOL$=STACK$(UPPER)
4130  UPPER=UPPER-1
4140  RETURN
```

Stacks may be implemented in other ways than shown here.

A PROGRAM USING A STACK

If we want to read a list of integers and output it in reverse order then we can program it this way, using a stack.

```
REM Output numbers in reverse order
REPEAT UNTIL there are no more numbers to read
        Read number N
        Push number N onto the stack
REPEAT UNTIL the stack is empty
        Pop number N from the stack
        Write number N
```

Here is the program:

```
010   REM PROGRAM REVERSE
020   REM READ NUMBERS AND OUTPUT IN REVERSE
030   DIM STACK(30)
040   UPPER=0
050   REM READ NUMBERS AND PUT ON STACK
060   PRINT "ENTER NUMBERS, END WITH -1"
070   COUNT=0
080   INPUT NUMBER
090   REM REPEAT UNTIL NUMBER= -1
```

```
100     IF NUMBER= -1 THEN GOTO 150
110         COUNT=COUNT+1
115         REM CALL SUBROUTINE ADD
120         GOSUB 4010
130         INPUT NUMBER
140         GOTO 100
150     REM END OF LOOP
160     REM POP NUMBERS FROM STACK AND OUTPUT
170     FOR I=1 TO COUNT
175         REM CALL SUBROUTINE SUBTRACT
180         GOSUB 4100
190         PRINT NUMBER
200         NEXT
210     END

4010    REM SUBROUTINE ADD(NUMBER,STACK,UPPER)
4020    REM PUSH NUMBER ONTO STACK
4030    UPPER=UPPER+1
4040    STACK(UPPER)=NUMBER
4050    RETURN

4100    REM SUBROUTINE SUBTRACT(NUMBER,STACK,UPPER)
4110    REM POP NUMBER OFF STACK
4120    NUMBER=STACK(UPPER)
4130    UPPER=UPPER-1
4140    RETURN
```

Notice that here the stack is a stack of numbers rather than names; it is dimensioned to have a maximum length of 30 elements. The value of UPPER is initialized to zero before the stack is used.

QUEUES

Another specialized type of list is a *queue.* For it, entries are made at the end of the list, deletions are made from the beginning. Rather than search for the end of the list each time an entry is made, it is usual to have a pointer indicating the last entry. Queues involve using things in a manner referred to as, "First in first out" (FIFO) or, "First come first served" (FCFS). This is the usual way for a queue waiting for tickets at a box office to operate.

A queue is not as easy as a stack to implement using an array. It is always growing at one end and shrinking at the other. If an array is used,

when the growth reaches the maximum limit of the array, we start it at the beginning again. Here is a queue of users of a computer waiting for service. We have a maximum of 8 elements. Five people are in the queue. The next person to be served is named GREEN.

FIRST 6	LAST 2
QUEUE$(1)	DUTCHISON
QUEUE$(2)	JOHNSTON
QUEUE$(3)	-
QUEUE$(4)	-
QUEUE$(5)	-
QUEUE$(6)	GREEN
QUEUE$(7)	LINNEMANN
QUEUE$(8)	JACOBS

Here are procedures used to ENTER or LEAVE this queue. We will use a constant named MAXQUEUE whose value is 8 so we can easily change the maximum size of the queue. Before using these procedures the queue can be initialized to be empty by setting FIRST to 1 and LAST to MAXQUEUE.

```
3010  REM SUBROUTINE ENTER(NAME$,QUEUE$,LAST)
3020  REM PLACE NAME$ AT END OF QUEUE
3030  LAST=LAST+1
3040  IF LAST>MAXQUEUE THEN LAST=1
3050  QUEUE$(LAST)=NAME$
3060  RETURN

3100  REM SUBROUTINE LEAVE(NAME$,QUEUE$,FIRST)
3110  REM REMOVE NAME$ FROM HEAD OF QUEUE
3120  NAME$=QUEUE$(FIRST)
3130  FIRST=FIRST+1
3140  IF FIRST>MAXQUEUE THEN FIRST=1
3150  RETURN
```

We have used wrap around or modulo arithmetic in that we wanted FIRST and LAST to keep increasing until they reach MAXQUEUE and then to wrap back to 1. The IF statements in the ENTER and LEAVE procedures make sure that wrap around occurs.

Queues can be implemented by linked lists as well as by simple arrays. Queues are used in the programs called *operating systems* that operate multi-user computer systems. Different jobs requiring service are placed in different queues, depending on the demands they are making on the system's resources and the priority that they possess to be given service.

Also, in programs that simulate other systems such as factories, queues are maintained to determine the length of time jobs are required to wait to be served when other jobs are competing for the same production facilities.

TREES

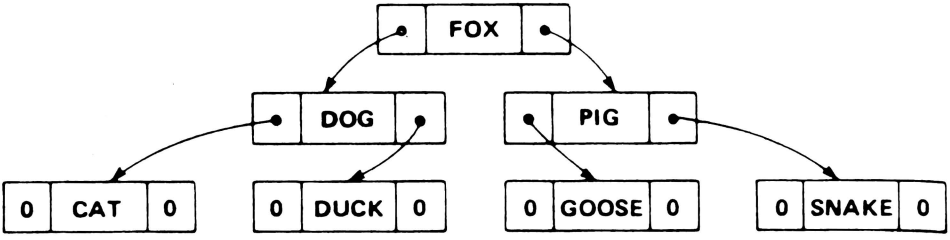
A linked list is an efficient way of storing a list that is changing with time, but it introduces an inefficiency in retrieval of information from the list. In Chapter 15 we saw that a binary search for an item in a list is much more efficient for long lists than a linear search. Unfortunately, there is no possibility of doing a binary search in a linked list; we must start at the beginning and trace our way through. There is no direct access to the middle of a linked list. It is for this reason that a more complicated data structure called a *tree* is used. We can get the efficiency of a binary search by having the elements linked into a *binary tree structure.*

To show how a binary tree is formed, we will look at the example of our list of names of animals:

```
          3  → CAT
     2       → DOG
                 DUCK
1                → FOX
                 GOOSE
                 PIG
                 SNAKE
```

To do a binary search we should begin in the middle. (We have added SNAKE to the list so the list has a middle entry.) If we are looking for the name CAT we find that CAT < FOX, so we then discard the middle entry and the last half of the list. The next comparison is with the middle entry of the remaining list, namely with DOG. Since CAT <DOG we eliminate the last half of the smaller list. By this time, we are down to one entry, which is the one we are looking for. It took three comparisons to get there. A linear search for CAT would, as it happens, have taken only 1 comparison. On the *average*, the binary search takes fewer comparisons than a linear search. A short list is not a good example for showing off the efficiency of binary searching, but it is much easier to write out all the possibilities.

We will now look at the binary tree that would be used to give the same searching technique. A diagram of it is on the next page.



Each data element in the tree structure consists of three parts, the NAME$ itself and two links that we designate as LEFTLINK and RIGHTLINK. The word FOX is in a special position in the tree, called the *root.* From FOX we have *branches* going to the left to DOG and to the right to PIG. In a sense, DOG is in the root position of a smaller tree, what we call the *left subtree* of the main tree. PIG is at the root of the *right subtree.* The words CAT, DUCK, GOOSE, and SNAKE are at the end of branches and are called *leaves* of the tree. All the data elements are in *nodes* of the tree; FOX is the *root node* and CAT is a *leaf node.* To search for an entry in a binary tree, we compare the element in the root node with the one we are seeking. If the root is the same, we have found it. If the root is larger we follow the LEFTLINK to the next entry; if smaller, we follow the RIGHTLINK. We are then at the root of a smaller tree, a tree with half as many entries as the original. The process is then repeated until the looked-for data is found.

Here is our tree structure as it might be stored in three arrays called NAME$, LEFTLINK, and RIGHTLINK. The variable ROOT holds the link to the root element. We have jumbled up the sequence to show that the actual order in the NAME$ array makes no difference. A zero link is used to indicate the end of a branch.

ROOT 4

	NAME$	LEFTLINK	RIGHTLINK
(1)	GOOSE	0	0
(2)	SNAKE	0	0
(3)	DOG	6	7
(4)	FOX	3	5
(5)	PIG	1	2
(6)	CAT	0	0
(7)	DUCK	0	0

Starting at ROOT, we find the root is in NAME$(4). LEFTLINK(4) leads us to NAME$(3) which is DOG. RIGHTLINK(3) leads us to NAME$(7) which is DUCK. You can see how it works. A tree structure is a *hierarchical structure* for data; each comparison takes us one *level* down in the tree.

ADDING TO A TREE

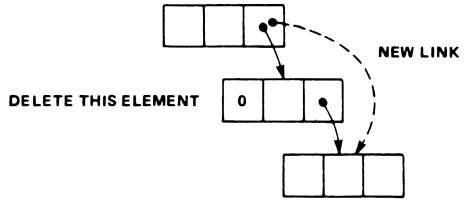
To add a data item to a tree structure we simply look for the element in the tree in the usual manner, starting at the root. If the element is not already in the tree, we will come in the search to a link that is null (zero). This is where the element belongs. In our example, if we want to add COW, we would start at FOX, then go to DOG, then to CAT. At this point we would want to follow the right link of CAT, but we find a zero. If we stored the new entry in NAME$(8), we would change RIGHTLINK(6) to 8 and set

NAME$(8)	LEFTLINK(8)	RIGHTLINK(8)
COW	0	0

As we add items to a tree, the tree becomes lopsided; it is not well balanced. Searching efficiency depends on trees being well balanced, so that in an information retrieval data bank using a tree structure, an effort should be made to keep the tree balanced. We started with a balanced tree and it became unbalanced by adding a new item. If a tree is grown from scratch using the method we have described for adding a new entry, it is unlikely to be well balanced.

DELETING FROM A TREE

Removing an entry from a tree is a more difficult operation than adding an entry. The same method is used to find the element to be deleted, but then the problem comes. It is not difficult if both links of the element to be deleted are zero, that is, if it is a leaf. We just chop it off and make the link pointing to it zero. If only one link is zero it is similar to an ordinary linked list and deletion is similar to that. We just bypass it.



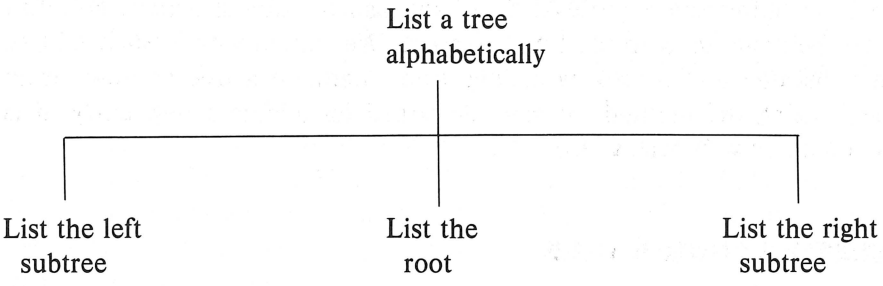
If neither link is zero in the element to be deleted, we must move another element into its position in the tree. In our original tree, if FOX is to be deleted, it must be replaced by an element that is larger then all other elements in the left subtree or smaller than all the elements in the right subtree. This means that either DUCK or GOOSE is the only possible choice. The one to be moved must be deleted in its present position before being placed in its new position.

Remember, in a linked structure, we never move a data item from its physical location in the data array we only change the links to alter its logical position.

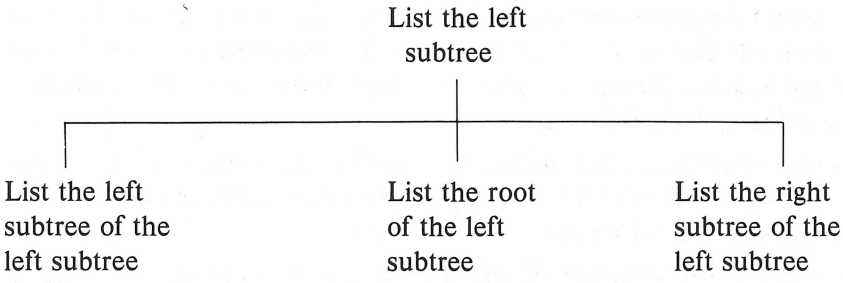
LISTING A TREE IN ORDER

Trees are used where searching and updating are the main activities. Sometimes we must list the contents of a tree. We must be systematic about it and be sure to list every node. We will show how to list it alphabetically.

An algorithm for listing a tree alphabetically can be written in this way:



We see that we have described our algorithm in terms of three parts. The middle part, "List the root," is easy, but the other two require us to, "List a tree." This is exactly what our problem is, to "List a tree." We have defined the solution to a problem in terms of the original problem. This kind of definition is called a *recursive* definition of a solution. It seems rather pointless, as if we were just going in a circle, but it really is not. The reason it is not pointless is that the tree we are attempting to list when we say, "List the left subtree," is a smaller tree than the original tree when we said "List a tree." When we try to list the left subtree we get this solution:



This time the left subtree of the left subtree has to be listed. It is smaller still. The algorithm is again repeated. Each application of the algorithm is on a smaller tree, until you reach a point where there is a zero link and there is *no* left subtree at all. Then the action of listing it is to do nothing: no tree, no listing. That is how recursive algorithms work. In programming terms, the algorithm calls itself over and over, each time to do a reduced task, until the task is easy to do.

In Basic a procedure may not call itself so to list a tree we would have to use a stack whose upper element would give the root of the particular subtree. Each point of return is added on top of the stack of other points of return. Finding the way back involves taking return points, one after the other, off this stack.

CHAPTER 17 SUMMARY

In previous chapters we have presented arrays which are data structures provided by Basic. In this chapter we showed how to build up new data structures using arrays. Some of these new data structures use links to give the ordering of data items. The link (or links) for a given item gives the array index of the next item. The following important terms were discussed:

Linked list - a linked sequence of data items. The next item in the list is found by following a link from the present item. The physical order of a collection of items, as given by their positions in an array, is different from their logical order, as given by the links.

Inserting into a linked list - a new data item can be inserted by changing links, without actually moving data items.

Deleting from a linked list - a data item can be deleted by changing links, without moving data items.

Available list - the collection of data elements currently not in use.

Stack - a data structure that allows data items to be added, or *pushed* on to one end and removed, or *popped*, from the same end. A stack does not require the use of links. A stack handles data items in a last-in-first-out (LIFO) manner.

Queue - a data structure that allows data items to be added at one end and removed from the other. A queue handles data items in a first-in-first-out (FIFO) manner.

Binary tree - a data structure in which each item or *node* has two links, a left link and a right link. The left link of a node locates another node and with it a subtree. Similarly, the right link locates a subtree. There is a unique beginning node called the *root*. If both links of a particular node are null, meaning they do not currently locate other nodes, then the node is called a *leaf*.

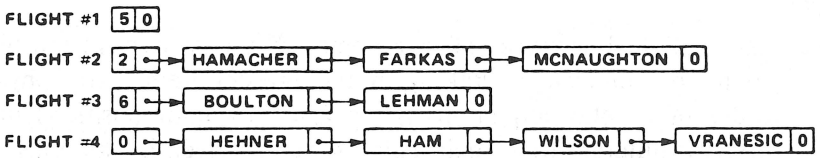
Recursive subroutine - a procedure that calls itself. Recursive subroutines are not possible in Basic.

CHAPTER 17 EXERCISES

1. The FLY-BY-NITE Airline company is computerizing its reservations system. There are four FLY-BY-NITE flights with the following capacities:

FLIGHT #1	5 seats
FLIGHT #2	5 seats
FLIGHT #3	8 seats
FLIGHT #4	4 seats

The information for passenger reservations is to be stored in a linked list. At some point during the booking period, the following diagram might represent the current passenger bookings.



The above diagram shows the first element in each list holding the number of seats remaining. Each succeeding pair of elements hold the name of a passenger and a link which either points to the next pair of elements or holds a 0 to indicate the end of the linked list.

In order to set up such a linked list system you will need an array, called FLIGHT, to contain the four "first" elements. Each of these elements holds two pieces of information, the number of seats remaining and the location of the first passenger.

The array called PASSENGER$ holds all the passengers. If all seats on all flights are taken, there will be CAPACITY=22 passengers. Hence PASSENGER$ will need a maximum of CAPACITY locations. Each element of the PASSENGER$ array contains the passenger's name and the corresponding element LINK contains the location of the next passenger, if any.

The pair of elements PASSENGER$(J) and LINK(J) correspond to each other. Before any events happen, the free locations must be linked together. Arrange that each LINK(J) contains a value J+1, except LINK(22) which contains a 0 as end of the list.

A variable AVAILABLE contains the location of the head of this chain of available locations. For the example given above, AVAILABLE contains 10 and PASSENGER$ and LINK could have these values:

PASSENGER$(1)	PASSENGER$(5)	PASSENGER$(9)
HAMACHER	LEHMAN	VRANESIC
LINK 6	LINK 0	LINK 0
PASSENGER$(2)	PASSENGER$(6)	PASSENGER$(10)
BOULTON	FARKAS	
LINK 5	LINK 7	LINK 11
PASSENGER$(3)	PASSENGER$(7)	.
HAM	MCNAUGHTON	.
LINK 8	LINK 0	.
PASSENGER$(4)	PASSENGER$(8)	PASSENGER$(22)
HEHNER	WILSON	
LINK 3	LINK 9	LINK 0

The reservation system is to accept four types of transactions:

Type 1 is a request for a reservation. The data entry contains the code word RES, name of the passenger, and the flight number.

Type 2 is a request to cancel a reservation. The data entry contains the code word CAN, name of passenger, and the flight number.

Type 3 is a request to display the number of seats remaining on a specified flight. The data entry contains the code word SEATS and a flight number.

Type 4 is a request to display a passenger list for the flight indicated. The data entry contains the code word LIST and a flight number.

Each type of transaction is to be handled by a subroutine. Here are descriptions of the subroutines:

ADD(WHO$,NUMBER). Adds passenger WHO$ to flight NUMBER. If that flight is filled, a message is displayed to that effect. ADD uses an element in PASSENGER$ and LINK and must update AVAILABLE.

CANCEL(WHO$,NUMBER). Cancels the reservation made in the name of WHO$ on flight NUMBER. The element in PASSENGER$ and LINK is returned to the free storage pool. AVAILABLE must be updated.

INFO(NUMBER). Displays the number of seats remaining on flight NUMBER.

DISPLAY(NUMBER). Displays a passenger list for flight NUMBER.

The data should simulate a real reservation system in that input of type 1, 2, 3, 4 should be intermixed. It would seem reasonable to assume that most cancellations would be made by persons holding reservations. However, people being what they are, you should not assume too much. In order to get your system off the ground, several reservation entries should be first.

Write and test each subroutine as a main program before putting the subroutines together. Write DISPLAY first and call it from ADD or CANCEL to help in debugging. Be sure to test "odd" situations as well as the obvious ones.

2. The INSERT subroutine in this chapter assumes that there are always enough elements to hold all items. Modify it to output an error message when there is not enough room for a new item. The DELETE subroutine assumes that the item to be deleted is actually in the list. Modify it to output an error message when the item to be deleted is not in the list.

Chapter 18

SCIENTIFIC CALCULATIONS

Most of the applications that we have discussed so far in this book are connected with the use of computers in business or in the humanities. We do business applications on computers because of the large numbers of each calculation that must be done. A single payroll calculation is simple, but if a company has a lot of employees, computer processing of payroll is warranted. Computers were originally developed with scientific and engineering calculations in mind. This is because many scientific and engineering calculations are so long that it is not practical to do them by hand, even with the help of a pocket calculator.

Often the scientific laws describing a physical situation are known in the form of equations, but these equations must be solved for the situation of interest. We may be designing a bridge or aircraft or an air-conditioning system for a building. A computer can be used to calculate the details of the particular situation.

Another important use of computers in science is to find equations that fit the data produced in experiments. These equations then serve to reduce the amount of data that must be preserved. Science as a word means knowledge. The object of scientific work is to gather information about the world and to systematize it so that it can be retrieved and used in the future. There is such a large amount of research activity now in science that we are facing an *information explosion.* We have talked about retrieving information from a *data bank* and computers will undoubtedly help us in this increasingly difficult and tedious job. But the problem of *data reduction* is of equal importance.

In this chapter we will try to give some of the flavor of scientific calculations, but we will not be including enough detail for those people who will need to work with them. We will give only an overview of this important use of computers.

EVALUATING FORMULAS

To solve certain scientific problems we must substitute values into formulas and calculate results. For example, we could be asked to calculate the distance traveled by a falling object after it is dropped from an airplane. A formula that gives the distance in meters traveled in time t seconds, neglecting air resistance, is

$$d = 4.9t^2$$

Here the constant 4.9 is one-half the acceleration due to gravity. Here is a program to compute the distance at the end of each second of the first 10 seconds after the drop:

```
010    REM PROGRAM FALL
020    REM OUTPUT TABLE OF DISTANCE FALLEN VERSUS TIME
030    TIME=0
040    REM LABEL TABLE
050    PRINT "TIME";TAB(16);"DISTANCE"
060    FOR I=1 TO 10
070        TIME=TIME+1
080        DISTNCE=4.9*TIME^2
090        PRINT TAB(4);TIME;TAB(16);DISTNCE
100        NEXT
110    END
```

The output for this program is

```
TIME            DISTANCE
   1              4.9
   2              19.6
   3              44.1
   4              78.4
   5              122.5
   6              176.4
   7              240.1
   8              313.6
   9              396.9
  10              490
```

This example outputs a table of values of distance for different times. Calculation of tables is an interesting and historic scientific use of computers. Scientific calculations are usually done using real values. In the output the distances and times are given with six digits in the fraction part, one digit to the left and five to the right of the decimal point. Not all these digits are *significant* the constant 4.9 in the formula is only expressed with two digits.

We must realize then that only about two digits of the distance traveled are significant.

The calculations are carried out in the computer keeping 6 digits, but this does not imply that they are meaningful. Even if the constant in the formula were entered to 6-digit precision, we would not necessarily have 6 significant digits in the answer. Because computers represent real numbers only to a limited precision, there are always what are called numerical errors. These are not mistakes you make but are inherent in the way that real numbers are represented in the computer. When two real numbers are multiplied, the product is rounded off to the same precision as the original numbers no more digits in the product would be significant. As calculations proceed, the rounding process can erode the significance even of some of the digits that are maintained. We usually quote numerical errors by saying that a value is, for example,

$$19.25 \pm 0.05$$

This means that the value could be as high as 19.30 or as low as 19.20. If the error were higher, say 0.5 instead of 0.05, then the values could range between 19.75 and 18.75. In this case the fourth digit in the value is certainly not significant, and you would say instead that the value was

$$19.2 \pm 0.5$$

Or we might round it off instead of truncating the insignificant digit, and write

$$19.3 \pm 0.5$$

The estimation of errors is an important job that is done by *numerical analysts.* If you are doing numerical calculations, you should be aware of the fact that answers are not exact but have errors.

PREDECLARED MATHEMATICAL FUNCTIONS

Scientific calculations require mathematical functions that are not commonly used in business calculations. For many of these functions, programs have already been written for Basic; they are predeclared in the interpreter. For example, suppose for our falling-body calculation we wanted to compute the times when the body reached different distances. To calculate the time, given the distance, we use this form of the same formula:

$$t = \text{the square root of } (d/4.9)$$

Now we need to be able to calculate a square root. This can be done by using the built-in function for square root, which is called SQR. We would write in the program:

```
TIME = SQR(DISTNCE/4.9)
```

Other predeclared functions available to Basic for scientific calculations are connected with trigonometry. They include SIN and COS. These give the values of the sine and cosine, when the argument of the function is in radians. ATN(X) gives the angle in radians whose tangent is X. The natural logarithm is obtained by using LOG, the exponential by using EXP.

GRAPHING A FUNCTION

Frequently a better understanding of a scientific formula can be had if you draw a graph of the function. In the first example of this chapter we evaluated a function at regular intervals. It is possible to use these values to plot a graph on the screen. We could, for instance, plot a distance-time graph for the falling object. We will show a way to plot a graph on the screen without using the APPLESOFT II graphics.

When you draw a graph of X versus Y you usually make the X-axis horizontal and have the Y-axis vertical. The values of X, which is the independent variable, increase uniformly the corresponding values of Y are obtained by substituting X into the function Y=f(X). When we plot a graph in text mode the lines of output text are uniformly spaced, so we will use the distance between lines to represent the uniform interval between the Xs. This means that the X-axis will be vertical and the Y-axis horizontal.

We represent the Y-value corresponding to the X of a particular output line by displaying an asterisk in the position that approximates its value. We use 31 columns to represent the range of Ys. If the lowest Y-value that we must represent is YSMALL and the highest is YBIG, then the 31 output positions must represent a range of

```
YRANGE = YBIG - YSMALL
```

To find the position for a value Y we compute an integer variable YPLACE from

```
YPLACE=INT(30*(Y-YSMALL)/YRANGE+1)
```

The value 1 is added to put YSMALL in the first position. We are assuming YRANGE is not zero so that we can divide by it. To form a string of characters for display, we replace the blank in the YPLACE position of a string of blanks by an asterisk. We first create a string variable called BLANKS$ that holds a string of blanks. The line of characters to be displayed, we call YLINE$. We put an X-axis on our graph if it is in the proper range. To do this we place a vertical bar (or exclamation mark) in the line of blanks and call the new variable BASICLINE$. The axis is

placed in the position where a zero value of Y would be placed. The axis does not appear at all if 0 is less than YSMALL or greater than YBIG. Here is the program segment for forming the BASICLINE$. Here instead of 31 output positions we assume a number given by a constant WIDTH. Then it can be easily changed if we want a wider or narrower graph.

```
BASICLINE$=BLANKS$
ZEROPLACE=INT((WIDTH-1)*(0-YSMALL)/YRANGE)+1)
IF(ZEROPLACE>=1) AND (ZEROPLACE<=WIDTH) THEN
    BASICLINE$=LEFT$(BLANKS$,ZEROPLACE-1)+"!"+
      MID$(BLANKS$,ZEROPLACE+1)
```

For each line we set YLINE$ to BASICLINE$ with an asterisk in the proper position. This is done by

```
YLINE$=LEFT$(BASICLINE$,YPLACE-1)+"*"+
  MID$(BASICLINE$,YPLACE+1)
```

We do not display a Y-axis, but we list the X-values corresponding to each line opposite the line.

A SUBOUTINE FOR PLOTTING GRAPHS

Here is the complete subroutine for plotting a graph from N pairs of values of X and Y stored in arrays of those names. The values of X are uniformly spaced. The actual names of the variables to be plotted will be given as arguments XNAME$ and YNAME$. We will call a subroutine to find YBIG and YSMALL. It will be called MINMAX.

```
6010   REM SUBROUTINE GRAPH(X,Y,N,XNAME$,YNAME$)
6020   REM PLOT A GRAPH IN TEXT MODE
6030   REM LOCAL VARIABLES:BLANKS$,WIDTH,BASICLINE$,
6040   REM YLINE$,YBIG,YSMALL,YRANGE,ZEROPLACE,
6050   REM YPLACE,I
6060   WIDTH=31
6070   REM FIND RANGE OF Y TO BE PLOTTED
6080   REM CALL MINMAX SUBROUTINE
6090   GOSUB 6500
6100   YRANGE=YBIG-YSMALL
6110   REM FORM STRING OF WIDTH BLANKS
6120   BLANKS$=""
6130   FOR I=1 TO WIDTH
6140       BLANKS$=BLANKS$+" "
6150       NEXT
6160   REM PLACE X-AXIS MARK IN BASICLINE
```

```
6170   ZEROPLACE=INT((WIDTH-1)*(0-YSMALL)/YRANGE+1)
6180   IF ZEROPLACE>=1 AND ZEROPLACE<=WIDTH THEN
         BASICLINE$=MID$(BLANKS$,1,ZEROPLACE-1)+"!"+
         MID$(BLANKS$,ZEROPLACE+1)
6190   REM LABEL GRAPH
6200   PRINT "GRAPH OF ";YNAME$;" VERSUS ",XNAME$
6210   PRINT "SMALLEST VALUE OF ";YNAME$;SPC(2);YSMALL
6220   PRINT "BIGGEST VALUE OF ";YNAME$;SPC(2);YBIG
6230   REM PREPARE AND OUTPUT LINES OF GRAPH
6240   FOR I=1 TO N
6250      YPLACE=INT((WIDTH-1)*(Y(I)-YSMALL)/YRANGE+1)
6260      YLINE$=MID$(BASICLINE$,1,YPLACE-1)+"*"+
            MID$(BASICLINE$,YPLACE+1)
6270      PRINT X(I);TAB(5);YLINE$
6280      NEXT
6290   RETURN

6500   REM SUBROUTINE MINMAX(Y,N,YBIG,YSMALL)
6510   REM FIND SMALLEST AND BIGGEST OF N ELEMENTS OF Y
6520   REM LOCAL VARIABLES: J
6530   YSMALL=Y(1)
6540   YBIG=Y(1)
6550   FOR J=2 TO N
6560      IF Y(J)<YSMALL THEN YSMALL=Y(J)
6570      IF Y(J)>YBIG THEN YBIG=Y(J)
6580      NEXT
6590   RETURN
```

USING THE GRAPH SUBROUTINE

We now give the program that can be used to plot the function of x,

$$y = x^2 - x - 2$$

between the values x=-2 and x=3. We plot it at intervals of x that are 0.2 wide. There are 26 points in all. Here is the program:

```
010    REM PROGRAM CURVE
020    REM PLOT THE FUNCTION Y=X^2-X-2
030    DIM X(50),Y(50)
040    N=26
050    SPACING=0.2
060    REM COMPUTE VALUES FOR X AND Y ARRAYS
070    FOR I=1 TO N
```

```
080       X(I)=-2+(I-1)*SPACING
090       Y(I)=X(I)^2-X(I)-2
100       NEXT
110    XNAME$="X"
120    YNAME$="Y"
130    REM CALL GRAPH SUBROUTINE
140    GOSUB 6010
150    END
```

The graph as displayed requires more than the 24 lines that you have on the screen so that as the bottom half is being displayed the image on the screen will scroll up. You can stop the scrolling by pressing CTRL-C. If you want the scrolling to continue type CONT. If you run this example you will notice that as the graph crosses the x-axis the exclamation mark is replaced by an asterisk. It crosses twice, at

$x = -1.00$ and at
$x = 2.00$

We say that $x=-1$ and $x=2$ are the roots of the equation

$$x^2-x-2=0$$

The function (x^2-x-2) becomes zero at these values of x. This graphical method is one way of finding the roots of an equation. We will look later in this chapter at another way of finding roots that is numerical rather than graphical.

FITTING A CURVE TO A SET OF POINTS

In the last sections we have seen how to compute a set of points of corresponding X and Y values from a formula and then to plot a graph of these points. In some scientific experiments we measure the value of a variable Y as we change some other variable X in a systematic way. The results are displayed by plotting X and Y. If there is a theory that relates the values of X to Y in a formula or equation, then we can see how well the results fit the theoretical formula.

One way would be to compute the values of Y for each X from the formula. The measured values could be called Y(experimental) and the calculated ones Y(theoretical). The differences between corresponding values

Y(experimental) - Y(theoretical)

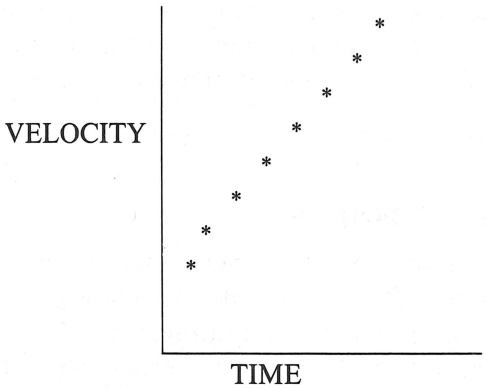
are called *deviations* of experimental from theoretical values.

We have spoken so far as if it were possible to compute the proper

theoretical value that corresponds to each experimental value. This is the case if the formula has no other variable in it. Frequently there are other variables in the formula that can change. For example, here is the formula for V, the velocity of an object at time T, given that its initial velocity is VINITIAL and its acceleration is A.

$$V = VINITIAL + A*T$$

If we measured the velocity of an object that has a uniform acceleration we could plot a graph between V and T. Theoretically, the graph should be a straight line, but the experimental points are scattered. It is possible to draw a line by eye that is placed so that the deviations of points from the line are small. Since some deviations, V(experimental)-V(theoretical), are positive and some negative their sum might be small even though individual deviations were large. To get a good fit we minimize the sum of the squares of the deviations rather than the sum of the deviations. The squares of the deviations are always positive. We choose as the best straight line the one that makes the sum of the squares of the deviations the least.



This is called *least-squares fitting* of a curve (here a straight line) to experimental points. This process can be done very efficiently by a computer. Most computer installations provide standard procedures for least-squares fitting, so that scientists do not have to write their own.

Sometimes no theoretical curve is known. We can still fit our data to an equation. We choose an equation that has a form resembling our data. If there is no theory we say it is an *empirical* fit, meaning that it is an equation based on the observations.

SOLVING POLYNOMIAL EQUATIONS

The graph that we plotted as an example was of the function

$$y = x^2 - x - 2$$

This is a *polynomial* function of x. The places where the graph crosses the x-axis are the *roots* of the equation

$$x^2 - x - 2 = 0$$

This is a second-degree equation since the highest power of the unknown x is the second power. It is a *quadratic equation.* There are general formulas for the roots of a quadratic equation. For the equation

$$ax^2 + bx + c = 0$$

the two roots x1 and x2 are given by the formulas

$$x1 = (-b + \sqrt{b^2 - 4ac})/(2a) \quad \text{and}$$

$$x2 = (-b - \sqrt{b^2 - 4ac})/(2a)$$

Most students of mathematics know these formulas. If the quantity $(b^2 - 4ac)$ inside the square root sign is positive, all is straightforward. If it is negative, then the formula requires us to find the square root of a negative number, and we say the roots are *complex.* This means, in graphical terms, that the curve does not cross, or touch, the x-axis anywhere. It is either completely above or completely below the x-axis. There is no use looking for values of x where the function is zero.

Here is a subroutine for finding the roots of a quadratic equation.

```
2010    REM SUBROUTINE ROOTS(A,B,C)
2020    REM PRINT ROOTS R1 AND R2 OF EQUATION
2030    REM A*X^2+B*X+C=0
2040    REM LOCAL VARIABLES: TEST,R1,R2,ROOT
2050    TEST=B^2-4*A*C
2060    REM IF TEST>=0 THEN...ELSE
2070    IF TEST>=0 THEN GOTO 2110
2080        REM DO ELSE
2090        PRINT "ROOTS ARE COMPLEX"
2100        GOTO 2160
2110        REM DO THEN
2120        ROOT=SQR(TEST)
2130        R1=(-B+ROOT)/(2*A)
2140        R2=(-B-ROOT)/(2*A)
2150        PRINT "ROOTS ARE ";R1;SPC(2);R2
2160    REM END OF SELECTION
```

2170 RETURN

In this subroutine the formulas for finding the roots do not provide values that are accurate under various circumstances. For example, if R1 is nearly zero because B and ROOT are very close in value, a better approximation to it can be obtained by working out R2 and using the assignment:

```
R1=C/(A*R2)
```

to compute R1. This relationship holds in general so it can always be used. Can you see why it is true?

For equations that are polynomial in x of degrees higher than two, the method for finding the roots is not as easy. For an equation of degree three or four, there is a complicated formula for the roots. For larger degrees there are no formulas and we must look for the roots by a *numerical method.*

The secret of any search is first to be sure that what you are looking for is in the right area, then to keep narrowing down the search area. One method of searching for roots corresponds to the binary search we discussed in Chapter 15. First we find two values of x for which the function has different signs. Then we can be sure that, if it is continuous, the graph will cross the x-axis at least once in the interval between these points. The next step is to halve the interval and look at the middle. If there is only one root in the interval, then in the middle the function will either be zero, in which case it is the root, or it will have the same sign as one of the two end points. Remember they have opposite signs. We discard the half of the interval that is bounded by the middle point and the end with the same sign and repeat the process. After several steps we will have a good *approximation* to the location of the root. We can continue the process until we are satisfied that the error, or uncertainty, in our root location is small enough. There is no point in trying to locate it more accurately than the precision with which the numbers are stored in the computer. A numerical analyst could determine the accuracy of the calculated answer.

SOLVING LINEAR EQUATIONS

Computers are used to solve sets of linear equations. If we have two unknowns, we must have two equations to get a solution. We can solve the set of equations

```
x-y=10
x+y=6
```

to get the result x=8, y=-2. To solve the equations we first eliminate one of the unknowns. From the first equation we get

x=y+10

Substituting into the second eliminates x. It gives

(y+10)+y=6 or 2y=-4 or y=-2

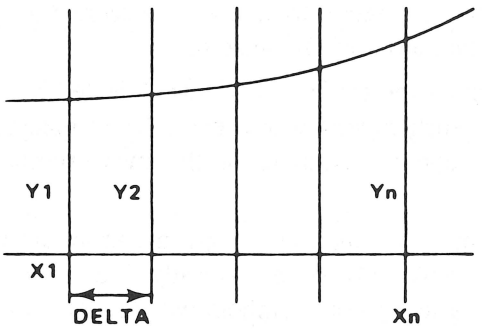
Then substituting back gives

x=-2+10 or x=8

This process of elimination can be carried out a step at a time for more equations in more unknowns. Each step lowers the number of unknowns by one and the number of equations by one. A computer program can be written to perform this job, and can be used to solve a set of linear equations. What we must provide is the coefficients of the unknowns and the right-hand sides of the set of equations. A common method is called the *Gauss elimination method.*

COMPUTING AREAS

Another numerical method that is relatively easy to understand is the calculation of areas by the *trapezoidal method.* Suppose we have a curve of y=f(x) and we want to find the area between the curve and the x-axis and between lines at x=X1 and x=Xn.



We will divide the distance between X1 and Xn into intervals of size DELTA. In the drawing we have shown four intervals. The area of the first section, thinking of it as a trapezoid, is

(Y1+Y2)*DELTA/2

The total area under the curve is approximated by the sum of all the trapezoids. The total area of the trapezoids is

```
(Y1+Y2)*DELTA/2+(Y2+Y3)*DELTA/2+...
                (Y(n-1)+Yn)*DELTA/2
```

If we factor out DELTA the formula becomes

```
((Y1+Yn)/2+Y2+Y3+... +Y(n-1))*DELTA
```

This is half the sum of Y1 and Yn plus the sum of the other Ys multiplied by the width of the trapezoids. As DELTA is made smaller, the sum of the areas of the trapezoids comes closer and closer to the area under the curve. It is a better and better approximation. There is, however, a limit to the accuracy that can be obtained, due to the precision of the real numbers. Here is a program segment to compute the area if the Ys are stored in an array:

```
SUM=(Y(1)+Y(N))/2
FOR I=2 TO N-1
    SUM=SUM+Y(I)
    NEXT
AREA=SUM*DELTA
```

CHAPTER 18 SUMMARY

This chapter has given an introduction to the use of computers in scientific calculations. Generally, these calculations are done using real numbers. The scientist needs to know the accuracy of the final answers. The answers may be inaccurate because of:

Measurement errors - the original data was collected by measuring physical quantities, such as length or speed. These measurements can never be perfect and an estimate of the measurement error should be made.

Round-off errors by the computer - a given computer stores real numbers with a particular precision, typically 6 decimal digits of accuracy. Calculations using real numbers will be no more accurate than the number of digits of precision provided by the computer. They could even be less accurate due to the cumulative effect of round-off. (Note: sometimes a programming language allows the programmer to choose between "single-precision" real, giving typically 6 digits of accuracy and "double-precision" real, giving typically 14 digits accuracy.)

Truncation errors in repeated calculations - some calculations, such as searching for the roots of a polynomial equation, produce approximations that are successively closer to the exact answer. When the repeated calculation stops, we have a truncation error, which is the

difference between the final approximation and the exact answer (ignoring errors due to measurement and round-off).

The number of digits of accuracy in a particular answer is called its number of *significant figures.* The scientist needs to know that the computer produces a particular answer with enough significant figures for his purposes.

Basic provides predeclared functions that are useful in solving scientific or mathematical problems. The function SQR takes the square root of a non-negative number. The functions SIN, COS, TAN and ATN operate on or return angles in radians. LOG takes the natural logarithm of a number, and EXP raises e to a specified power.

This chapter presented the following typical scientific and mathematical uses of computers.

Evaluating formulas - a computer can produce tables of numbers, for example tables of navigational figures used on sailing boats.

Graphing functions - a computer can plot a particular function using the display screen. Graphs can be obtained using the APPLESOFT II graphics package or by using the ordinary text mode display.

Fitting a curve to a set of points - data points from an experiment can be read by a program and used to determine an equation (a curve) that describes the data.

Solving polynomial equations - a polynomial equation such as

$$x^3+9x^2+6x-23=0$$

can be solved by a program that reads the coefficients (1, 9, 6 and -23).

Solving linear equations - a set of equations such as

$$2x + 9y = 7$$

$$10x - 4y = 2$$

can be solved by a program that reads the coefficients of the unknowns (2, 9, 10 and -4) and the right sides of the equations (7 and 2).

Areas under curves - a program can find the area under a given curve by using the heights of the curve at many points. Essentially, the program slices the area into narrow strips and adds up the areas of the strips. This process is sometimes called numerical integration or quadrature.

CHAPTER 18 EXERCISES

1. One jet plane is flying 1083.7 kilometers per hour another jet plane, chasing it from behind, is flying 1297.9 kilometers per hour. What is the relative speed of the second plane, that is, how fast is it catching up to the first plane? The speed of the first plane is known to an accuracy of ±5 km/hr and the speed of the second is known to an accuracy of ±0.5 km/hr. How accurately can we calculate the relative speed? How many significant figures are there in the first plane's speed, the second plane"s speed and the relative speed?

2. Use the graphing procedure given in this chapter to plot the function SIN(X) for X varying from zero to three in steps of one tenth.

3. Use the graphing procedure given in this chapter to plot the function X*SIN(X) for X varying from 0 to 12 in steps of 0.25.

Appendix 1:

SYNTAX OF BETTER BASIC FOR THE APPLE

The Applesoft Basic syntax is given in the *Applesoft II, Basic Programming Reference Manual* published by Apple Computer Inc. What is given here is the syntax of the Basic that is presented in this book. In this Basic the use of certain remark lines is part of the language convention. The GOTO statement of Basic is used in a very controlled way.

In describing the syntax we use this notation:

[item] means the item is optional
{item} means the item can appear zero or more times

Note that these brackets are not part of the Basic language. They are part of a *meta-language* used to describe the syntax of Better Basic. The symbol] is used in Basic as the prompt symbol but is never written in a Basic program. These two kinds of brackets are called *meta-symbols.* The vertical line | is a meta symbol meaning "or".

When presenting the syntax, items written in *upper case* letters, for example

REM or END

denote either keywords of Applesoft Basic or are words like

PROGRAM or SUBROUTINE

which we are using, by convention, to make our programs more understandable. These items appear in programs exactly as they are presented. We say they are terminal symbols; they will be part of the final program. Items written in *lower case* letters, for example,

statement

denote one of a class of elements; each such item must appear in a subsequent (or perhaps preceding) definition. We call them *syntactic elements.* The syntax is defined in such a way that all syntactic elements can ultimately be substituted for by *terminal symbols* since the final program contains only terminal symbols.

Many Basic program statements can be executed in an *immediate execution mode.* This book stresses programming rather than using the computer as a direct calculator so that no reference will be made in this syntax to whether or not a statement can also be executed in the immediate execution mode.

A *character* is one of the following:

letter
digit
special character

A *letter* is:

A |B |C |D ... |Z

A *digit* is:

0 |1 |2 |3 ... |9

A *special character* is:

! |# |$ |% |& |' |" |> |(|* |: |- |

@ |+ |; |? |/ |) |. |< |, |^ |blank

An *expression* is one of the following:

arithmetic expression
string expression

An *arithmetic expression* is one of the following:

integer
real
arithmetic variable
[+ |-] (expression)
expression arithmetic operator expression
built-in arithmetic function
FN user-defined function

An *arithmetic operator* is:

+ |- |* |/ |^

(Note: the operator * means multiply; ^ means to the power.)

A *variable* is one of:

arithmetic variable
string variable

An *arithmetic variable* is one of:

name
name (index {,index})

(Note: name cannot be, or contain, a Basic reserved word.)

A *string expression* is one of the following:

literal
string variable
built-in string function
string expression + string expression

(Note: the operation + used here means to concatenate the values of string expressions.)

A *string variable* is one of the following:

name$
name$ (index {,index})

(Note: name cannot be, or contain, a Basic keyword.)

A *literal* is:

"{character}"

A *name* is:

letter {letter | digit}

An *index* is:

arithmetic expression

Digits are:

digit {digit}

An *integer* is:

[+ | -] digits

A *real* is one of the following:

[+ |-] digits. [digits] [E [+ |-] digits]
[+ |-] [digits]. digits [E [+ |-] digits]
[+ |-] digits E [+ |-] digits

(Note that a real has a decimal point or explicit exponent.)

A *line number* (linenum) is:

digits

(Note: the largest permitted line number is 63999.)

An *array declaration* is:

DIM array name (integer {,integer}) {,array name(integer {,integer}) }

A *constant declaration* is:

constant name = expression

(Note: the type of the constant name, that is, string or arithmetic, must match the type of the expression.)

A *condition* is one of the following:

comparison
(condition)
NOT condition
condition AND condition
condition OR condition

A *comparison* is one the the following:

expression < expression
expression > expression
expression = expression
expression <= expression
expression >= expression
expression <> expression
(<> means not equal to)

A *function declaration* is:

DEF FN name (parameter name) = arithmetic expression

(Note: line numbers in programs and subroutines must be in ascending sequence, as indicated.)

A *subroutine declaration* is:

```
linenum     REM SUBROUTINE name [(parameter{,parameter})]
linenum     REM purpose of subroutine
linenum     REM LOCAL VARIABLES: [variable{,variable}]
linenum     statement
{linenum    statement}
linenum     RETURN
```

A *program name* is:

name

(Note: a program name *may* be or contain a Basic keyword. First 30 characters used for identification.)

A *program* is:

```
linenum     REM PROGRAM program name
linenum     REM purpose of program
{linenum    array declaration}
{linenum    constant declaration}
{linenum    function declaration}
[linenum    error control statement]
linenum     statement
{linenum    statement}
linenum     END
            {subroutine declaration}
            {data line}
```

A *statement* is one of the following:

remark
assignment statement
output statement
input statement
subroutine call statement
counted loop construct
infinite loop construct
if...then statement
repeat until construct
if...then...else construct
case construct
disk handling construct

display mode statement
low-resolution graphics statement
high-resolution graphics statement
tracing control statement
error control statement
poke statement
terminal statement

A *remark* is:

REM {character}

(Note: a remark may be appended to any single line statement if preceded by a colon.)

An *assignment statement* is one of:

arithmetic variable = arithmetic expression
string variable = string expression

An *output statement* is one of the following:

PRINT
PRINT output item {separator output item} [separator]
PRINT [format specifier;] output item {;[format specifier;] output item}
HOME
VTAB arithmetic expression
HTAB arithmetic expression
PR#0
PR#1

An *output item* is:

expression

A *separator* is:

, |;

A *format specifier is:*

TAB (character position)
SPC (number of spaces)

A *character position* is:

integer expression

An *input statement* is one of:

INPUT variable {,variable}
READ variable {,variable}

(Note: the INPUT statement expects the input from the keyboard unless it is preceded by a READ disk handling statement without a subsequent PRINT D$ statement. The READ statement accepts input from data lines.)

An *input item* is one of the following:

{character} (without commas or colon)
literal
integer
real

A *data line* is:

linenum DATA input item {,input item}

A *subroutine call* statement is:

GOSUB line number of first line of subroutine

(Note: in all constructs each line has its own line number.)

A *counted loop construct* is:

FOR variable = expression TO expression [STEP expression]
statement
{statement}
NEXT

An *infinite loop construct* is:

REM INFINITE LOOP
line A statement
{statement}
GOTO line A

An *if...then statement* is:

IF condition THEN statement {:statement}

(Note: the statements must be non-construct statements.)

A *goto statement* is:

GOTO linenum

(Note: we do not use this statement except in the if...then statement or in a construct.)

A *return statement* is:

```
        RETURN
```

(Note: used only as the last statement of a subroutine.)

A *repeat until construct* is:

```
                REM REPEAT UNTIL condition
line A          IF condition THEN GOTO line B
                    statement
                    {statement}
                    GOTO line A
line B          REM END OF LOOP
```

An *if...then...else construct* is:

```
                REM IF condition THEN...ELSE
                IF condition THEN GOTO line A
                    [REM DO ELSE
                    statement
                    {statement}]
                    GOTO line B
line A              REM DO THEN
                    statement
                    {statement}
line B          REM END OF SELECTION
```

A *case construct* is:

```
                REM CASE CONSTRUCT
                ON case number GOTO line A, line B, ... line N
line A              statement
                    {statement}
                    GOTO line Z
line B              statement
                    {statement}
                    GOTO line Z
                    ...
line N              statement
                    {statement}
line Z          REM END OF CASE
```

(Note: a case construct may use GOSUB instead of GOTO)

A *disk handling construct* is:

PRINT D$; "disk-handling statement"
PRINT D$; "READ file name, R"; record number
PRINT D$; "WRITE file name, R"; record number

(Note: D$ must be assigned the value CHR$(4).)

A *disk handling statement* is one of the following:

OPEN file name
CLOSE file name
LOCK file name
UNLOCK file name
READ file name
WRITE file name
RUN file name

A *file name* is:

letter {character}

(Note: there is a maximum of 30 characters which may include spaces.)

A *display mode statement* is one of the following:

TEXT
GR
HGR
FLASH
INVERSE
NORMAL

A *low-resolution graphics statement* is one of the following:

COLOR = integer expression having value between 1 and 15
VLIN arithmetic expression 1, arithmetic
    expression 2 AT arithmetic expression 3
HLIN arithmetic expression 1, arithmetic
    expression 2 AT arithmetic expression 3
PLOT arithmetic expression 1, arithmetic expression 2

(Note: the arithmetic expression must have values inside low-resolution graphics window.)

A *tracing control statement* is one of:

TRACE
NOTRACE

A *high-resolution graphics statement* is one of the following:

HCOLOR = integer expression having value between 0 and 7
HPLOT [arithmetic expression 1, arithmetic expression 2]
{TO arithmetic expression 3, arithmetic expression 4}

(Note: the arithmetic expressions must have values inside high-resolution graphics window.)

An *error control* statement is:

ONERR GOTO line number of first statement
in error handling subprogram

(Note: an error handling subprogram may return control to the statement where the error occurred by the resume statement.)

A *resume* statement is:

RESUME

A *poke* statement is:

POKE machine location, number to be stored

A *terminal* statement is one of the following:

END
STOP

The following constructs are consistent with the better Basic philosophy but were *not* in fact used in the book. The first is an alternative conditional loop called the WHILE...DO. It has the definition.

A *while...do construct* is:

```
          REM WHILE condition DO
line A    IF NOT (condition) THEN GOTO line B
              statement
              {statement}
              GO TO line A
line B    REM END OF WHILE
```

The other construct is an alternative form of IF...THEN...ELSE. It has this definition.

An *if...then...else construct* is:

```
                REM IF condition THEN...ELSE
                IF NOT (condition) THEN GOTO line A
                    REM DO THEN
                    statement
                    {statement}
                    GOTO line B
line A              REM DO ELSE
                    statement
                    {statement}
line B          REM END OF SELECTION
```

If there is no ELSE this reduces to:

```
                REM IF condition THEN
                IF NOT (condition) THEN GOTO line B
                    REM DO THEN
                    statement
                    {statement}
line B          REM END OF SELECTION
```

Appendix 2:

APPLESOFT BASIC COMMANDS (DOS COMMANDS)

command	*use*
NEW	clears memory
RUN [program name]	"zeroes" variables, executes program
LOAD program name	clears memory and loads program
SAVE program name	records program on disk
LIST [line number][—line number]	lists lines of program in memory

or
LIST line number -
(Note: a comma can also be used instead of a minus sign in LIST commands)

ctrl-C	stops execution or listing
CONT	continues execution or listing
ctrl-X	cancels line being typed
DEL line number[,line number]	deletes lines of program in memory
TRACE	sets to trace programs executed

NOTRACE	turns off trace
PR#1	causes program output to be printed
PR#0	returns output to screen
CATALOG	lists file names on disk
EXEC text file name	executes text file
LOCK file name	locks disk file
UNLOCK file name	unlocks file
RENAME present file name, new file name	changes name of file
INIT identification program name	initializes diskette
DELETE file name	deletes file from disk
esc A	moves cursor to right
esc B	moves cursor to left
esc C	moves cursor down
esc D	moves cursor up

(Note: cursor moves one position when escape key is pressed, then released and letter is pressed.)

HTAB position	moves cursor along line to position
VTAB line number	moves cursor to line number
left arrow (←)	backspaces and rubs out character
right arrow (→)	moves cursor right accepting character moved over

Appendix 3:

RESERVED WORDS IN APPLESOFT BASIC

&, ABS, AND, ASC, AT, ATN
CALL, CHR$, CLEAR, COLOR=, CONT, COS
DATA, DEF, DEL, DIM, DRAW
END, EXP
FLASH, FN, FOR, FRE
GET, GOSUB, GOTO, GR
HCOLOR=, HGR, HGR2, HIMEM:, HLIN, HOME, HPLOT, HTAB
IF, IN#, INPUT, INT, INVERSE
LEFT$, LEN, LET, LIST, LOAD, LOG, LOMEM:
MID$
NEW, NEXT, NORMAL, NOT, NOTRACE
ON, ONERR, OR
PDL, PEEK, PLOT, POKE, POP, POS, PRINT, PR#
READ, RECALL, REM, RESTORE, RESUME, RETURN, RIGHT$, RND, ROT=, RUN
SAVE, SCALE=, SCRN(, SGN, SHLOAD, SIN, SPC(, SPEED=, SQR, STEP, STOP, STORE, STR$
TAB(, TAN, TEXT, THEN, TO, TRACE
USR
VAL, VLIN, VTAB
WAIT
XPLOT, XDRAW.

Of these the following were not discussed in this book:

&, CALL, CLEAR, DRAW, FRE, GET, HGR2, HIMEM:, IN#, LOMEM:, POP, POS, RECALL, ROT=, SCALE=, SCRN(, SHLOAD, USR, WAIT, XPLOT, and DRAW.

Consult the Applesoft II Reference Manual for details of these.

Appendix 4:

BUILT-IN APPLESOFT BASIC FUNCTIONS

For these functions the arguments are numerical. The result is numerical.

SIN(x) - sine of x radians

COS(x) - cosine of x radians

TAN(x) - tangent of x radians

ATN(x) - angle in radians whose tangent is x

INT(x) - largest integer less than or equal to x

ABS(x) - absolute value of x

SGN(x) - -1 if $x < 0$, $+1$ if $x > 0$, 0 if $x = 0$

SQR(x) - square root of x

EXP(x) - e to the power x

LOG(x) - natural logarithm of x

RND(1) - random number between 0 and 1 from series

RND(-x) - sets seed for series of random numbers

PEEK(x) - number stored in machine location x

For these functions the values and/or arguments are strings.

LEN(string) - integer; length of string

MID$(string,start,length) - - substring of string starting at position "start" for length of "length"

LEFT$(string,length) - substring of leftmost length characters of string

RIGHT$(string,length) - substring of rightmost length characters of string

CHR$(x) - character whose ASCII code is x

ASC(character) - ASCII code of character

VAL(string) - numeric value of string of digits

STR$(x) - the string equivalent of number x

These functions read the game controls

PDL(n) - a number between 0 and 255 of game paddle n

PEEK(n - 16287)- value exceeds 127 if button of game control n pressed

This function causes a click on the speaker.

PEEK(-16336).

Appendix 5:

ERROR MESSAGES AND CODES

When an error occurs in the execution of a program, execution is terminated, and control is returned to the command level as shown by the display of the prompt signal] and flashing cursor.

If, however, the program includes the error control statement then control is transferred to an error handling subprogram. The error code number is stored in memory location 222. The line number at which the error occurred can be determined by the expression:

PEEK(218)+PEEK(219)*256

Error messages displayed are of the form

? error description ERROR IN line number

These error messages are detailed in the Applesoft Basic Programming Manual. Most are self explanatory.

Here are some more general ones.

SYNTAX ERROR	bad punctuation, bad variable name, illegal character
STRING TOO LONG	maximum string length is 255 character
ILLEGAL QUANTITY	improper function argument
UNDEF'D STATEMENT	transfer to non-existent line number
TYPE MISMATCH	assignment whose left- and right-hand sides do not match in type

Here are some error codes.

code	*Error type*
0	NEXT without FOR
16	Syntax
22	RETURN without GOSUB
42	Out of data to read
53	Illegal quantity
69	Overflow
77	Out of memory
90	Undefined statement
107	Bad subscript (array index)
120	Array already dimensioned
133	Division by zero
163	Type mismatch
176	String too long
191	Formula too complex
224	Undefined function
254	Bad input
255	Control-C interrupt

Appendix 6:
ASCII CHARACTER CODE

DEC = ASCII decimal code
CHAR = ASCII character name
n/a = not accessible directly from the APPLE II keyboard

DEC	CHAR	KEY(s)	Hex	DEC	CHAR	KEY(s)
0	NULL	ctrl @	20	32	SPACE	space
1	SOH	ctrl A		33	!	!
2	STX	ctrl B		34	"	"
3	ETX	ctrl C		35	#	#
4	ET	ctrl D		36	$	$
5	ENQ	ctrl E	25	37	%	%
6	ACK	ctrl F		38	&	&
7	BEL	ctrl G		39	'	'
8	BS	ctrl H **or** ←		40	(	(
9	HT	ctrl I	29	41	)	)
10	LF	ctrl J	2A	42	*	*
11	VT	ctrl K	2B	43	+	+
12	FF	ctrl L	2C	44	,	,
13	CR	ctrl M **or** return	2D	45	-	-
14	SO	ctrl N	2E	46	.	.
15	SI	ctrl O	2F	47	/	/
16	DLE	ctrl P	30	48	0	0
17	DC1	ctrl Q		49	1	1
18	DC2	ctrl R		50	2	2
19	DC3	ctrl S		51	3	3
20	DC4	ctrl T		52	4	4
21	NAK	ctrl U **or** →		53	5	5
22	SYN	ctrl V		54	6	6
23	ETB	ctrl W		55	7	7
24	CAN	ctrl X		56	8	8
25	EM	ctrl Y		57	9	9

26	SUB	ctrl Z	58	:	:
27	ESCAPE	ESC	59	;	;
28	FS	n/a	60	<	<
29	GS	ctrl shift-M	61	=	=
30	RS	ctrl ^	62	>	>
31	US	n/a	63	?	?

64	@	@
65	A	A
66	B	B
67	C	C
68	D	D
69	E	E
70	F	F
71	G	G
72	H	H
73	I	I
74	J	J
75	K	K
76	L	L
77	M	M
78	N	N
79	O	O
80	P	P
81	Q	Q
82	R	R
83	S	S
84	T	T
85	U	U
86	V	V
87	W	W
88	X	X
89	Y	Y
90	Z	Z
91	[	n/a
92	\	n/a
93	]	] (shift-M)
94	^	^
95	_	n/a

INDEX

A, 35
ABS, 158
Accumulator, 13, 18
ADD, 13
Address, 12
Address calculation, searching by, 184
Algorithm, 132, 139
  developing of, 134
  recursive, 239
Alphabetic information, 115
Alphabetic order, 119
Alphameric, 115
Alphanumeric, 115
Analyst, numerical, 245
AND, 67, 99
Animation, 170, 177
APPEND, 204, 217, 220
Applesoft Basic, 53
Applesoft II, 7
  Basic, 5, 6
  Basic Programming Reference Manual, 257
Area, computation of, 253
Area under curve, 255
Argument, 156
Arithmetic calculation, 1
Arithmetic expression, 22, 29, 258
Arithmetic operator, 258
Arithmetic unit, 9, 13
Arithmetic variable, 259
Array, 106
  bounds, 112
  as data structure, 110
  declaration of, 105, 112, 260
  element of, 106
  index, 112
  of records, 217
  two-dimensional, 107
Artificial intelligence, 145
ASC, 124, 127
ASCII character code, 124, 275
Assignment, 60
  statement, 47, 262
Asterisk, 23
AT, 165
ATN, 158
Audio tape, 11
Available list, 239

Basic, 6
  Applesoft II, 5
  interpreter, 59, 151
  keyword, 46
  reading of, 150
Basic Programming Reference Manual, 164
Batch processing of program, 52, 210
Beep, 169
Bin, 185
Binary digit, 10
Binary search, 182, 188, 252
  algorithm for, 182
Binary system, 10
Binary tree, 240
  structure, 235
Bit, 10, 17
Block, basic building, 93
Body of the ELSE, 77
Body of loop, 69, 92

Body of program, 26
Boolean (logical) operator, 67
Boot, 24
Boot the system, 39
Bottom-up test, 197
Branch, 132
  three-way, 77
  of tree, 111
Branching, 75
Brownian movement, 170
Bubble sort, 143, 188
Bucket, 185
Bug, 195
Building block, 153
Built-in function, 158, 161, 271
Byte, 10, 17

Calculation, arithmetic, 1
Calculator, stored program, 15
CALL, 164
Calling a subroutine, 160
Capturing a program or subroutine as a text file, 219
Caret, 23
CASE construct, 80, 86, 267
Case-like construct, 156
CATALOG, 35, 71
Catalogue listing, 219
Central processing unit, 9, 17
Chaining programs, 219
Changing a character, 42
Character, 10, 19, 28, 258
  code, 124
  correct, 36
  deletion of, 38, 42
  identifier, 116
  prompt, 24
  reading and outputting of, 116
  replacement of, 38
  special, 19
  string, 22, 115
  variable, 45
CHR$, 124, 127
Clicks, 169
CLOSE, 204, 218
Code, 10
  character, 124
Column, 107
Colon, 20
COLOR, 164, 176
Color for low-resolution graphic, 164
Comma, 34
Command, 5
  language, 5, 7
  system, 5
  for text files, 219
Comment, 136
Common error, 58
Comparison, 84, 260
Complex roots, 251
Complexity, control of, 93, 136
Computation of area, 253
Concatenation, 120, 126
Condition, 2, 66, 84, 260
  for stopping, 70
Conditional loop, 68, 87
  flow chart for, 94
Constant, 21, 54
  declaration of, 260
  definition of, 60
Construct, CASE, 80, 86
Construct, IF...THEN...ELSE, 80, 85
CONT, 35
Control, 14
  key, 35
  register, 14
  TAB, 27
  unit, 18
Control Flow, tracing of, 82, 87
Controlling complexity, 93

Coordinate, 165
Copying diskette, 39
Correct character, 36
Correction of line, 36
Correctness of program, 4, 6
COS, 158, 173
Counted FOR loop, 65, 83
Counted loop construct, 263
CPU, 9, 17
CTRL-C, 35, 70
CTRL-X, 37
Cursor, 26, 37
    movement of, 37, 41
Curve, fitting of, 249, 255

DATA, 52, 60
Data, 2
    bank, 237, 243
    input of, 50
    line, 263
    processor, 2
    reduction, 243
    retrieval, 179
    structure, 223
        array, 110
        choosing of, 133
    type of, 45
Dataset, 204, 217
Debugging, 195, 200
Decimal digit, 10
Decimal point, 21
Decimal system, 10
Decision, 2
Decision criterion, 2
Declaration, of array, 105
DEF, 157
Defensive programming, 192
DEL, 36
DELETE, 38, 204
Deletion, 36
    of character, 38, 42
    of line, 36, 41
    from linked list, 229, 239
    from tree, 237
Deviation, 249
Desk-checking a program, 192
Dice, 159
Digit, 20, 258, 259
    binary, 10
    decimal, 10
DIM, 105
Dimension of array, 105
Direct execution, 199
Disk, 11
    handling, 204
        construct, 265
        statement, 204, 218, 265
    operating system, 5, 39
    track of, 11
Disk file, handling, 204
Diskette, 12
    copying of, 39
    initialization of, 33, 40
Display mode statement, 265
Displaying a catalogue, 41
Divide-and-conquer strategy, 140, 153
Division, 23
Documentation, 62, 154
DOS, 5, 24, 32, 33, 39, 213
DOS command, 268
Drawing lines, 166
Drive, 34
Dummy argument, 157
Dummy entry, 70, 71
Dummy name, 119, 180

Editing a program, 36
Efficiency, assessing of, 136
Efficiency, of sorting method, 187
Efficient algorithm, 137
Element swapping, 133

Empirical fit, 250
END, 26
End condition, 195
End-of-file, 85
    error, 215
    marker, 70
END OF LOOP, 69
Engineering calculation, 243
Envelope, of diskette, 34
Equal sign, 47
Equation, root of, 249
Equation, quadratic, 251
Error message, 58, 273
Error, 167
    code, 274
    common, 58
    control statement, 266, 273
    location of, 216
    numerical, 245
    in program design, 198
    of semantics, 18, 58
    of syntax, 16, 18, 46, 57, 62
Escape D, 37
Escape key, 37
Evaluation, of formula, 244, 255
EXEC, 212, 215, 219
Executable statement, 47
Execution, tracing of, 49
Exhaustive testing, 5
Exit from loop, 92, 101
EXP, 158
Exponent, 21
Expression, 22, 258

False, 67
FCFS, 233
Field, 24, 29
    of record, 179
FIFO, 233
File, 204
    listing of, 207
    maintenance of, 208, 219
    name of, 265
    random access to, 210
    of records, 188
    in secondary memory, 204
    sequential, 204
File actions, monitoring of, 207
Fit, empirical, 250
Fitting of curve, 249, 255
Fitting, least-squares, 250
FLASH, 164
Floppy disk, 12
Flow chart, 93, 101
    for conditional loop, 94
    for IF...THEN...ELSE, 93
FN, 157
FOR loop, 66, 108
Form of statement, 16
Formal language, 145
Format specifier, 262
Formatted output, 29
Formula, evaluation of, 244, 255
Fraction, 21
    part of, 21
Function, 157
    built-in, 158, 161
    declaration of, 260
    definition of, 161
    graphing of, 246, 255

GOSUB, 155
GOTO, 68, 92, 101
GOTO statement, 263
GR, 164, 176
Game control, 168, 177
Gauss elimination, 253
Graphics, 163
    high-resolution, 171, 177
Graphing a function, 178, 246, 255

HCOLOR, 172, 178
HGR, 177
HLIN, 167
HLIN...AT, 177
HOME, 25, 125
HPLOT, 172
HPLOT...TO, 178
HTAB, 38
Halving of interval, 252
Hardcopy, 15
Hash-coded, 185
Hierarchical structure, 237
High-level language, 3, 6, 16
High-resolution graph mode, 163, 177
High-resolution graphic, 171, 177
High-resolution graphics statement, 266
Histogram, 149

I/O, 9
ID program, 34
Identifier, 14, 45, 60
  character, 116
Identification of diskette, 40
Identification message, 34
IF condition THEN statement, 68
IF statement with multiple conditions, 100
IF...THEN...ELSE, 76
  construct, 80, 85, 264, 267
  flow chart for, 93
IF...THEN statement, 263
Immediate execution mode, 258
Index, 105, 259
Indexed FOR loop, 66
INFINITE LOOP, 70, 75
Infinite loop construct, 263
Information, 2
  alphabetic, 115
  coded, 10, 18
  explosion, 243
  processor, 2
  retrieval, 237
INIT, 34, 40
Initial program, 33
Initialization, 69, 72
  of diskette, 33, 40
  of variable, 46
Input, 9, 50, 60
  of data, 50
  item, 263
  statement, 262
  reading of, 70
Input/output terminal, 15
Input terminal, keyboard, 1
Input and output units, 15, 17
Inserting, 42
  character, 42
  into linked list, 224, 226, 239
Instruction, machine language, 14
Instruction pointer, 14
Instructions, repetoire of, 1
INT, 158
Integer, 21, 46, 259
  constant, 21, 29
Intelligence, artificial, 145
Interactive processing, 210
Interactive system, 15
Interpretation, 18
Interpreter, 1, 6, 151
  Basic, 3, 5, 59
Interval halving, 252
INVERSE, 164
Invoking a function, 161

Joining strings, 120
Jump, 14

Key, 24
  control, 35
  escape, 37
  left-arrow, 36
  return, 24
  right-arrow, 37

Keyboard, 9, 17
input terminal, 1
Keyword, 26, 62
in Basic, 46, 270

Label, 26
Labelling of graphics, 178
Labelling of output, 55
Language, 1
command, 5, 7
formal, 145
high-level, 3, 6, 16
machine, 1
natural, 145
programming, 1, 6
Large scale integrated circuits, 12
Leaf node, 236
Least-squares fitting, 250
LEFT$, 122, 127
Left-arrow key, 36
LEN, 117, 126
Length, of string, 117
Length, of word, 11
Letter, 20, 258
Line, 26
correction of, 36
deletion of, 36, 41
number, 26, 260
Line-at-a-time printer, 15
Linear equation, solving of, 252, 255
Linear search, 179, 188
Link, 111, 224
Linked list, 223, 239
deleting from, 229, 239
inserting into, 224, 226, 239
maintenance of, 230
LIST, 35, 41, 213
List, 105
linked, 223
memory management with, 225
Listing of file, 207
Listing of program, 36, 41
Listing of tree, 238
Literal, 22, 29, 259
LOAD, 13, 36
Loading a program, 40
Local variable, 154, 160
LOCK, 38, 41
LOG, 158
Logical sequence, 224
Loop, 65, 83
body of, 69, 84
counted FOR, 65, 83
infinite, 70, 75
with multiple conditions, 98
nested, 95
phases of, 69
problems with, 95
structure of, 91
unconditional, 70, 85
Loudspeaker, 169
Lower case letter, 257
Low-resolution graph mode, 163, 176
Low-resolution graphics statement, 265
LSI, 12

Machine language, 1
instruction, 14
Machine, sequential, 15
Magnetic tape, 11
Maintenance of files, 208
Maintenance of linked list, 230
Mantissa, 21
Mark, punctuation, 19
Marker, end-of-file, 70
Matrix, 108
Measurement error, 254
Memory, 1, 9, 17, 45
location in, 48
management of, 225

with lists, 225
Merge sort, successive, 187
Merging, sorting by, 186
identification of, 34
prompt, 51
Message, 34
Meta-language, 257
Meta-symbol, 257
Microprocessor, 18
MID$, 121, 127
Misspelling, 57, 199
Mixed number, 21
Module, 153, 155
prefabricated, 139
Modular programming, 132, 139, 155
MON, 207, 218
Monitoring file actions, 207
Morse code, 10
Move cursor, 37
Movement of cursor, 38, 41
Multiple condition, 101
Multiplication, 23
Multiply-dimensional array, 112
Name, 259
Natural language, 145
Nested loops, 95
Nested statements, 101
NEW, 26
NEXT, 108
Node, 132
in tree, 111
NOMON, 208, 219
NORMAL, 164
NOTRACE, 83, 196
Null string, 116
Number, 20, 21
of line, 26
random, 159
rounding of positive, 50
Numerical analyst, 245
Numerical error, 245
Numerical variable, 45, 46
Numerical method, 252

ON, 80, 156
ON...GOSUB, 160
ONERR, 168, 170, 173, 216
ONERR GOTO, 220
OPEN, 204, 218
Operating system, 6
for disk, 5
OR, 67, 99
Order, alphabetic, 119
Origin of coordinates, 173, 177
Out-of-bounds, index, 112, 124
Output, 9, 24, 30
item, 262
labelling, 55
line of, 24
printed, 27, 30
statement, 262

PDL, 168, 177
PEEK, 59, 62, 167, 168
PEEK (218), 220
PEEK (222), 220
Physical sequence, 224
PLOT, 165, 177
Plotting a graph, 173
Plotting, line from point to point, 172
Plotting, a point, 165
Padded with blanks, 119
Paddle, 168
Palindrome, 151
Paragraphing the program, 81, 86
Parameter, 154
Parameters of subroutine, 160
Parameter-argument correspondence, 174
Parentheses, 22
Pause, 10
Payroll calculation, 243
Pointer, 111

instruction, 14
POKE, 59, 62, 163
POKE statement, 266
Polynomial, 252
Popped from stack, 231
Posting to file, 208
Power to printer, 28
PR, 27, 30
Precedence, rule of, 23, 29
Precision, 245
Predeclared function, 255
Prefabricated module, 139
Preparation of file, 206
PRINT, 24, 29, 199
PRINT D$, 205, 218
Printed output, 27, 30
Printer, line-at-a-time, 15
Probability, 159
Product of matrices, 113
PROGRAM, 26
Program, 6, 26, 261
  body of, 26
  correctness of, 4, 6, 193, 200
  design error in, 198
  desk-checking of, 192
  editing of, 36
  initial, 33
  listing of, 36, 41
  paragraphing of, 81, 86
  saved as text file, 213
  specification for, 191, 199
  testing of, 57
  translation of, 16
  well-structured, 131
  language, 1, 6
  modular, 132, 139
  structured, 2, 6
  style of, 193, 200
Programming, 1, 6
  defensive, 192
  habits of, 199
Prompt, 51
  character, 24
  message, 51
Proving program correct, 5
Pulse, 10
Punctuation mark, 19
Pushed onto stack, 231

Quadratic equation, 251
Question-answering system, 145
Question mark, 51
Queue, 233, 240
Quote, 25
Quotation marks, 117

Random access to file, 210
Random number, 159
Random-access file, 217
READ, 52, 60, 204, 218, 219
Read, 15
Reading, 70
  Basic, 150
  input, 70
Reading and outputting character, 116
Reading string, 117
Real, 260
  constant, 29
  form, 21
Recognition, of word, 146
Record, 179, 201, 217
  field of, 179
  file of, 188
  number, 210
Recording device, 11
Rectangle, 167
Recursive definition, 238
Recursive subroutine, 240
Reduction of data, 243
Reel, of tape, 12
Refinement, step-by-step, 131, 140
Register, control, 14

Relational, expression, 67
REM, 26, 53
Remark, 26, 52, 53, 60, 262
  use of, 141, 193, 200
RENAME, 38
REPEAT UNTIL, 66
  conditional loop, 68, 84
  construct, 264
Repeat, 37
Repetition, 42, 57, 65
Replacement of character, 38
Representative sample, 4
REPT, 37, 42
Reserved word, 46, 270
Reservation system, 240
RESTORE, 52
RESUME, 217, 220, 266
Retrieval, of data, 179
Return key, 24
RETURN statement, 264
RIGHT$, 122, 127
Right-arrow key, 37
RND, 159, 170
Root, of equation, 249
Root node, 236
Root, of tree, 111, 132
Rounded off, 245
Rounding, 61
  positive number, 50
Round-off error, 254
Row, 107
Rule of precedence, 23, 29
RUN, 26, 40
Running a saved program, 40

Sample, representative of, 4
SAVE, 35
Saving a program, 40
Scalar product of vector, 113
Scientific calculation, 243
Scientific notation, 21
Screen, 9, 17
Scroll, 35, 163, 176
Search, 179
  by address calculation, 184
  binary, 182
  linear, 179
  time for, 181
Secondary memory, files in, 204
Seed for series of pseudo-random numbers, 159
Selecting part of string, 121
Selection, 65 75
Semantic error, 18, 58
Serial number, 185
Separator, 262
Sequence, logical, 224
Sequence, physical, 224
Sequencing strings, 119
Sequential control, 75
Sequential file, 204, 217
Sequential machine, 15
SGN, 158
Shell sort, 143
Significant digit, 244
SIN, 158, 173
Size of window, 59
Slash, 23
Solution tree, growing of, 133
Solving, of linear equation, 252, 253
Solving, of polynomial equation, 255
Sort, shell, 143
Sorting, 133, 185
  algorithm, 143
  efficiency of method, 187
  by merging, 186
Sound on the computer, 169
Source program, 1
Speaker, 177
SPC, 25
Special character, 19, 258
Special condition, 196
Special symbol, 20

Specification for program, 191
SPEED, 35, 41, 208, 219
SQR, 158
Stack, 231, 232, 239, 240
Start up, 39
Statement, 20, 261
  assignment, 47
  executable, 47
Statistics, word, 148
Step-by-step refinement, 13, 140
Stopping condition, 70
Stopping program execution, 40
STORE, 13
Stored program calculator, 15
String, 115
  of characters, 115, 116
  expression, 259
  length of, 117
  reading of, 117
  variable, 259
Strings, 119
  comparison of, 126
  joining of, 120
  sequencing of, 119
Structure, 2
  for data, 223
  of loop, 91
  of nesting, 81
  of tree, 111
Structured programming, 2, 6
  goals of, 2
STR$, 126, 128
Subprogram, reasons for using, 159
SUBROUTINE, 153
Subroutine, 153, 160
  call statement, 263
  calling or invocation of, 155, 160
  declaration of, 261
Substring, 121
Subtree, 236
Successive merge sort, 187
Symbol, 19
Symmetric array, 107
Syntactic element, 257
Syntax error, 16, 18, 46, 57, 62
System, operating, 6
System command, 5

T, 204
TAB, 25
TAB control, 27
Table, 107
TAN, 158
Tape reel, 12
Teletype machine, 10
Terminal statement, 266
Terminal symbol, 257
Testing, 195, 200
  exhaustive, 5
  of program, 57
TEXT, 60, 165, 176
Text file, 204, 211, 219
  for execution, 212
Text mode, 163, 176
THEN body, 77
Three-way branch, 77
Top-down approach, 131, 137, 140
Top of stack, 231
TRACE, 82, 196
Tracing control flow, 82, 87
Tracing program execution, 49
Tracing control statement, 265
Track, of disk, 11
Transaction, 208, 241
Translation, of program, 16
Trapezoidal method, 253
Tree, 235
  deletion from, 237
  root of, 132
  structure, 111
Tree structure to problem solution, 132, 140
Trigonometry, 172

True, 67
Truncation, 61, 245
  error of, 254
Two-dimensional array, 107, 110
Type, 45, 60
  of data, 45
Typewriter, 15

Unconditional loop, 70, 85
UNLOCK, 38
Upper case letter, 257
User-defined function, 158, 161

VAL, 28, 126
Value of variable, 45
Variable, 45, 259
  character, 45
  numerical, 45, 46
VLIN, 166
VLIN...AT, 177
VTAB, 38, 42

Well-balanced tree, 237
Well-structured program, 131
WHILE...DO construct, 266
Window, 163, 176
  graphic, 171, 176
  size of, 59
Work habit, 192
Word, 11, 17
  address of, 12
  contents of, 13
  length of, 11
  recognition of, 146
  reserved, 46
  statistics for, 148
WRITE, 204, 218
Write, 15

X-coordinate, 165

Y-coordinate, 165

better BASIC for the apple

J.N.P. Hume
R.C. Holt

Now two nationally-recognized authors/computer professionals have combined their talents to give you a complete guide to programming your Apple® computer. All of the programs contained in this new book are structured programs—making it easy to write, enter, edit, rearrange, and run these programs on your own under the disk operating system (DOS) using Applesoft BASIC.

Everything you need to use your Apple® computer effectively in business and home applications is contained in this book. For someone who wants to get the most out of their computer, this handy guide is packed with practical examples ready to be put to use.

The Table of Contents includes:

- Introduction to Structured Programming
- The Computer
- Getting a Computer to Calculate and Output
- Editing and Saving Programs
- Variables, Constants, and Assignments
- Control Flow
- Structuring Control Flow
- Arrays
- Alphabetic Information Handling
- Structuring Your Attack on the Problem
- The Computer can Read English
- Subprograms
- Graphics
- Searching and Sorting
- Making Sure the Program Works
- Files and Records
- Data Structures
- Scientific Calculations

Cover design by Robin Judy

RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia 22090

0-8359-0466-0

Apple® is a registered trademark of Apple Computers, Inc.

Better BASIC for the Apple

Huma • Holt

Reston